IAN SINCLAIR

MEMOTECH COMPUTING

Memotech Computing

Memotech Computing

Ian Sinclair

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

British Library Cataloguing in Publication Data
Sinclair, Ian
Memotech computing.
1. Memotech (Computer)—Programming
I. Title
001.6'42 QA76.8.M4

ISBN 0-246-12408-3

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface

Of all the new computers that have been announced in the past few years, the Memotech range has aroused very great interest because of its excellent specification and moderate price. This book is intended as a guide for the new Memotech owner, and is intended to supplement the manual, not to replace it. For the beginner to computing, the Memotech offers the attractions of being straightforward to learn to program, with a high-quality keyboard that makes it much easier to acquire typing skills. A further attraction is that the machine is one that can be expanded as your needs develop. Many Memotech machines, however, will undoubtedly be bought by readers who have owned or used other computers, and who are even better placed to appreciate the advantages of the Memotech. This book is also designed to help these owners, who will discover that the BASIC of the Memotech, though similar to other BASICs, has many points of individuality which make it difficult to adapt to without guidance.

This book aims to provide the guidance which will be needed for beginner and second-time user alike, to progress with programming the Memotech. Particular attention has been paid to the graphics of this machine, because they are very extensive, and most rewarding to master. One complete chapter has been devoted to the unique NODDY system of text processing, which relieves the programmer of the chore of trying to arrange lines of text neatly in BASIC. The SOUND instruction has also been given a complete chapter to itself, as has the subject of sprites.

As always, a book of this kind is the result of teamwork. I am grateful to Richard Miles of Granada Publishing, who commissioned the book, produced the machine, and gave encouragement throughout. Sue Moore, also of Granada, performed the usual miracles of meticulous checking, and typesetters and printers combined to get this book to you as fast as paper could be shifted.

Ian Sinclair

Chapter One
Setting Up

By the time that you unpack it, you will have found that the Memotech is not the type of computer that will blow away in a wind. It is a large machine in a solid steel case, designed to stand up to a lot more use than the average chunk of plastic. The case contains the keyboard and all the computing circuits, but not the power supply. The power supply is a separate box which connects to the computer by a cable that ends in a six-pin plug. This plug carries a notch which will ensure that it fits one way round into the socket on the keyboard that is marked POWER. Make sure that the notch is uppermost, and take care over inserting the plug, because you can damage the pins if you use a lot of force. The socket is recessed, so that it's not easy to guide the plug into place. The machine is then ready whenever you connect a mains plug to the power supply. A computer is a more complicated device than a kettle or a toaster, however, and connecting a mains plug is just the start of getting the machine working its wonders for you.

The plug is connected as indicated in Fig. 1. 1. There are only two leads, one blue and the other brown, and the cable should be tightly clamped. The fuse should be a 3 amp type, not the 13 amp variety which usually comes with the three-pin plug. If you are accustomed to fitting plugs for yourself then the diagram should be enough to remind you of what is needed. If you don't want to have anything to do with mains supplies, then take the Memotech power supply box (you don't need to take the whole computer) along to an electrician and get a plug, with a 3A fuse, connected.

With that hurdle over, you are almost ready to work some Memotech magic, but you need the use of a TV receiver. A computer is a device which is arranged so as to send signals to a TV receiver, and unless you connect a TV receiver to the Memotech you won't be able to see what the Memotech is doing. It will still compute for you just as well, but you won't see what is going on.

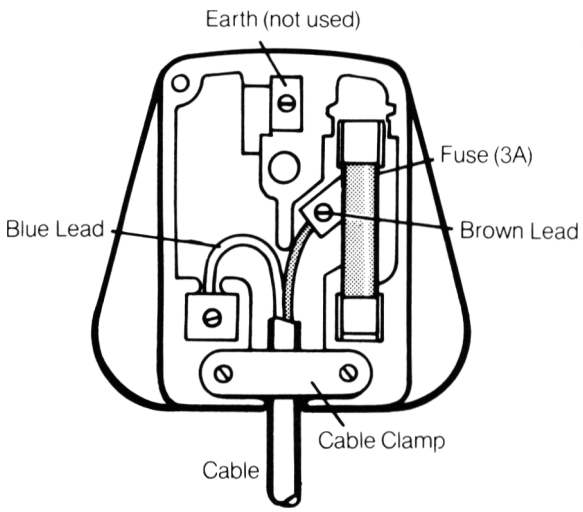


Fig. 1.1. The connections to a three-pin mains plug. Only the live and neutral leads are used. If you haven't done this sort of thing before, play safe and hand it to an electrician.

The Memotech comes with a TV cable ready to attach. There is a 'phono plug' at one end of this cable, and an aerial plug at the end of the lead, as Fig. 1.2 illustrates. The phono plug fits into the socket on the Memotech that is marked TV. You could, of course, simply plug the other end of this lead into the TV receiver, but a better option is to use the type of 2-to-1 adaptor that is illustrated in Fig. 1.3. This allows you to keep an aerial cable plugged in, and to connect or disconnect the Memotech as you wish without disturbing the TV receiver. It's useful if you have to share a colour TV with the family. It also saves wear on the aerial connector of the TV receiver itself. If you have a TV that you can reserve for use with the Memotech then

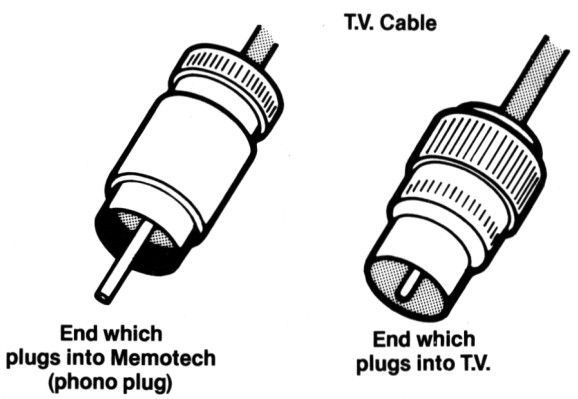


Fig. 1.2. The two different plugs on the TV cable.

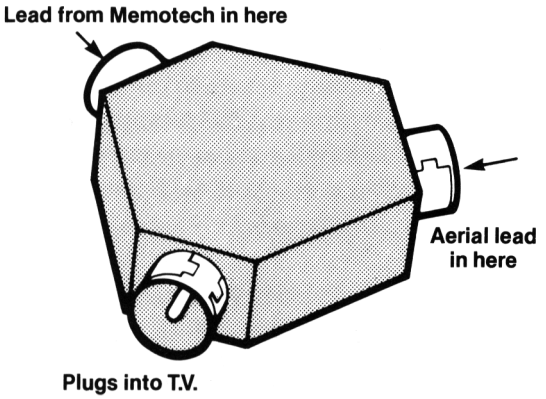


Fig. 1.3. A typical 2-to-1 adaptor for TV cables which you can buy at electrical stores.

you won't need this device which is sold in my local radio shop as a *Panda Pack*.

The TV that you use to display the Memotech's signals need not be a colour receiver, not to start with at least. The skills of programming a Memotech do not require you to see the results in colour until you come to the colour instructions of the Memotech in Chapter Seven. You can learn a great deal about computing using only a black and white portable, and even when you get to colour displays, you will see these as different shades of grey. An ordinary domestic TV, however, is not ideal for viewing the Memotech (or any other computer) signals. This is because the signals cannot be sent directly to the TV in the form that would give a clear picture. Instead, they have to be transmitted, using a miniature transmitter that is called a modulator. This is because most TV receivers cannot be safely connected to anything except by the aerial lead. Very much clearer pictures can be obtained by using what is called a *monitor*. This is a form of stripped-down TV which can't receive broadcast signals (no licence needed!), but which can be safely connected to the Memotech (and to a few other types of computers) to show high-quality pictures. Though you can't get *Dallas* on it, you can get very much better displays! Suitable monitors are available from computer shops. The Memotech can be adjusted so as to use either a black and white or a colour monitor, and if you intend to use either type, you should consult your Memotech dealer to make sure that the correct connections have been made inside the keyboard. If you can read a circuit diagram, the connections are the links that are labelled 'a' and 'b' on the PAL VIDEO BOARD diagram. Link 'b' should be

connected for use with a B/W monitor, and link 'a' for a colour monitor. Do *not* attempt this for yourself, however, unless you are familiar with circuit diagrams and printed circuit boards. If you buy a colour monitor, make sure that it has 'composite video' input – a monitor with only 'RGB input' *cannot* be used. In general, if the monitor can be used with a video recorder, it can also be used with the Memotech.

The big switch-on

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have around. When you are in full control of your Memotech you will need three mains sockets. Two of these will be for the Memotech and the TV receiver, but you will need one more for a cassette recorder. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of a three- or four-way socket strip with a cable and a plug (Fig. 1.4). This avoids a lot of what the famous advert calls 'spaghetti hanging out of the back'.

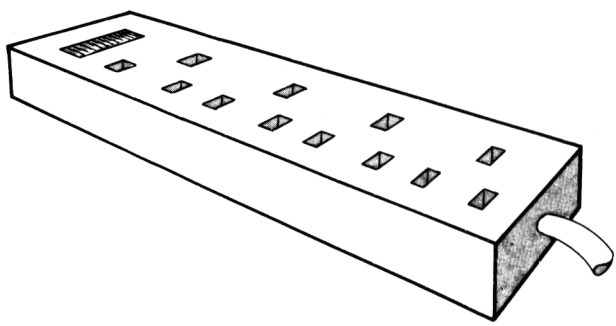


Fig. 1.4. A four-way socket strip which avoids the use of the old-style adaptors.

Don't rely on the old fashioned type of three-way adaptor – they never produce really reliable contacts. The Memotech has its own mains switch, so you can keep it plugged in if you like, but it's always better to unplug everything when you finish a computer session.

The next step, then, is to switch on the TV receiver and the Memotech. The power switch for the Memotech is on the side of the power supply box, and it lights up when it is switched on. The Memotech is capable of delivering sound signals, and these are

played through the loudspeaker of the TV so that you have full control over the volume. You can also send these signals out to a hi-fi system so that you can hear them at full volume, or record them, as you please. For connection to a sound system, there is a single phono socket, marked HI-FI next to the power socket at the back of the Memotech. Your Memotech dealer will be able to supply suitable connecting leads for this purpose, or you can buy leads from any audio dealer.

The next point is that a TV receiver has to be tuned to the signal from the Memotech. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked VCR it's unlikely that you will be able to get the Memotech tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the Memotech's signals. Figure 1.5 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Fig. 1.5(a). This is the type of tuning system that you find on black and white portables, and to get the Memotech's signal on the screen, you only have to turn the dial. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then start with the dial turned fully anticlockwise as far as it will go, and slowly turn it clockwise until you see the Memotech signal appear. If you turn the volume control up slightly so that you can hear the rushing noise of the untuned receiver, you will hear things go quiet as the Memotech signal appears. You may find that there is some reduction in the sound level as you tune to a local TV transmission, but you'll notice the difference. The Memotech doesn't give you the sound of *Dallas*!

What you are looking for, if the Memotech hasn't been touched since you switched it on, is the word 'Ready' on the screen. When you can see this word, turn the dial carefully, turning slightly in each direction until you find a setting in which the word is really clear. On a TV receiver, particularly a colour TV, the word may never be particularly clear, but get it steady at least and as clear as possible. If you are using a colour receiver, you will see the rest of the screen in a deep blue colour, and the 'Ready' in white or a very light blue.

The older types of colour and B/W TV receivers use mechanical push-buttons (Fig. 1.5(b)) which engage with a loud clonk when you push them. There are usually four of these buttons, and you'll need to use a spare one, which for most of us means the fourth one. Push this one fully in. Tuning is now carried out by rotating this button.

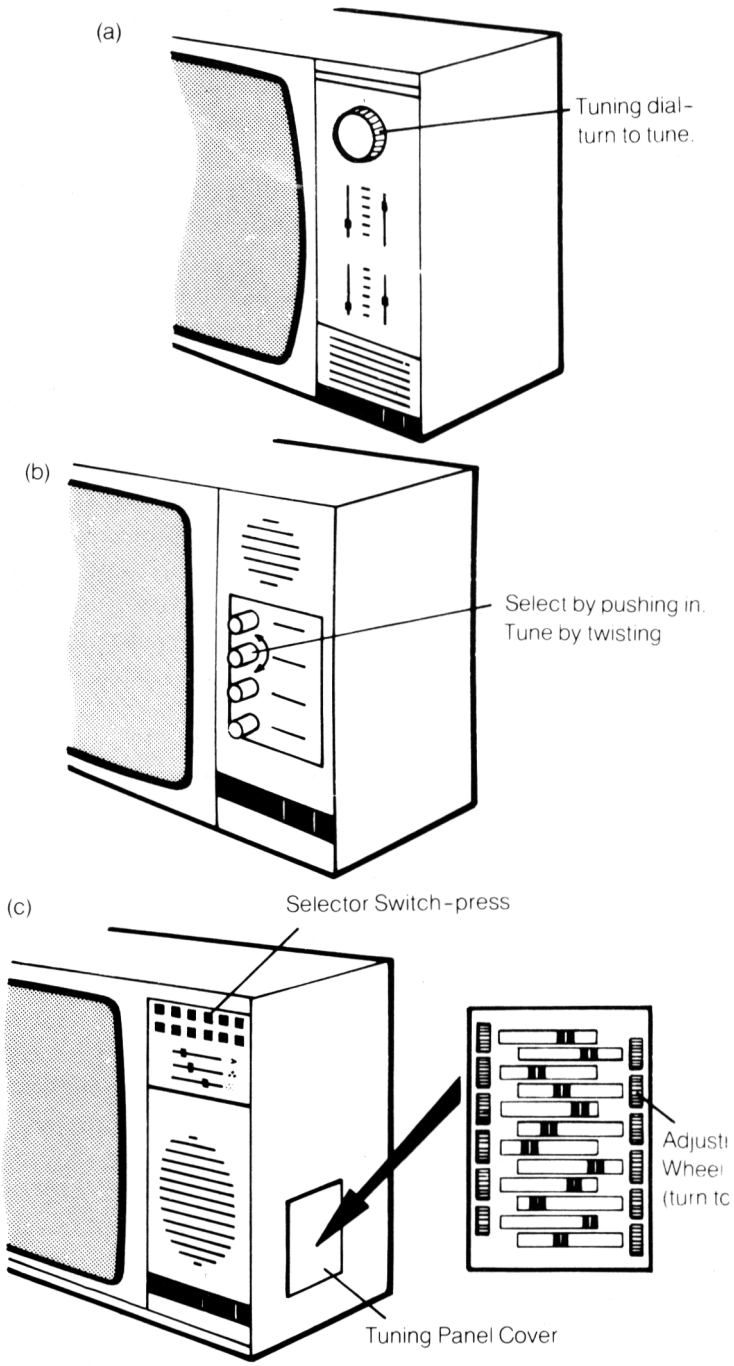


Fig. 1.5. TV tuning controls. (a) Single dial, as used on black and white portables, (b) four-button type, (c) the more modern touch-pad or miniature switch type.

Try rotating anticlockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the Memotech's signal during this time, you'll see and hear the same signs – the message on the screen and the reduction in the noise from the loudspeaker. If you've turned the button all the way anticlockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the Memotech signal at any setting, check the TV using an aerial in case there is something wrong with the tuning of the TV.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver (Fig. 1.5(c)). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen and silence from the loudspeaker. On this type of receiver, the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave it off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep re-tuning. Contrary to what a lot of dealers may tell you, not all TV receivers are suitable for use with a computer. The Memotech performs best with a Sony Trinitron receiver, but when I used it with a Philips portable, I could not get completely satisfactory results. The most common problem is a loud buzz or hum on the sound when the colour is correctly tuned. It's another good reason for using a colour monitor!

The mystery starts

Once you have achieved a tuned signal from your Memotech, the business of mastering the Memotech magic begins. To start with, you have the message 'Ready' shining at you from the screen. This means what it says, that the computer is waiting for a command from you. Some distance above the 'Ready', you'll see a flashing square. This is called the *cursor*. It is used as a marker, and when you press a key, a letter (a number, or whatever is marked on the key) will

appear at the position of the cursor. The cursor will then move across the screen to the next position. It's important to note at this point that nothing that you can do by pressing keys on the keyboard can possibly damage the Memotech – the worst you can do is to lose a program that was stored in the memory. You can, however, damage the Memotech by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on. *Always* switch off the computer, and everything that is connected to it when you insert or remove any of the plugs at the back.

Key-tapping time

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the Memotech. If we ignore the keys on the left and the right hand sides, most of the Memotech keys look like typewriter keys. The arrangement of letters and numbers is the same as that of a typewriter and if you've ever used one, particularly an electric typewriter, then you should be able to find your way round the keyboard of the Memotech pretty quickly.

There's one very noticeable difference, though. When you use a typewriter, pressing a letter key gives you a small letter (called *lower-case*), and pressing a letter key along with the SHIFT key produces a capital letter (called *upper-case*). On the Memotech keyboard, you will get upper-case (capitals) when you press letter keys by themselves, or when you use the SHIFT key along with the letter key. You can get lower-case (small letters) when you press letter keys after pressing the ALPHA LOCK key on the left-hand side of the keyboard. After pressing this key once, the keyboard of the Memotech behaves just like the keyboard of a typewriter. Commands that you give to the computer in typed form can be in either lower-case (small letters) or upper-case (capitals). You will be able to type capitals with the SHIFT key pressed. To return to the original system, you only have to press the ALPHA LOCK key again. Whatever the setting of the ALPHA LOCK key, the keys which show two symbols on them, like most of the number keys, will need the use of SHIFT. When you press one of these keys alone, you get the symbol that is marked on the lower part of the key. Using SHIFT along with the key gives the symbol on the upper part of the key.

As well as the ordinary typewriter keys, there are a number of special keys which are not found on any typewriter. At the left-hand

side of the two top rows of keys, for example, you will find the keys that are labelled ESC (escape) and CTRL (control). These are used in ways that are outlined in the manual – the names aren't really a good description of what they do. We'll look later at some of the most useful applications of these keys. There is also a pair of keys that are not marked in any way, one on each side of the space-bar, on the front of the keyboard. These are 'panic buttons' which when pressed *together* will return the control of the Memotech to you if it appears to have 'locked up' and refuses to obey instructions. Pressing just one of these reset keys has no effect, so you really have to intend to use them. The reason for making this difficult is that pressing these keys completely resets the computer, wiping out any program that happened to be running at the time. Another key, marked BRK (on the 9 key of the separate number keypad at the right-hand side of the main keyboard) will stop the Memotech from carrying out program actions, and return it to awaiting your next command. If the machine refuses to obey a command, then the key that is marked CLS (on the number keypad) will clear out the command. Finally, the set of eight keys at the extreme right-hand side of the keyboard comprises what are called the *programmed function* keys. Their uses are explained in the text, and also in Appendix C.

The most important of these special keys, however, as far as we are concerned at the moment, is the key that is marked RET. This is in the position of the carriage return key of an electric typewriter, but its action is not the same in all respects. Pressing the RET key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it.

If you are accustomed to using an electric typewriter, you will have to change some of your habits as far as this key is concerned. During the use of a typewriter, you would press the carriage return key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The RET key of the computer does rather more than this. If the material that you are typing into the Memotech takes more than one line on the screen, the machine will *automatically* select the next screen line for you. The RET key must *not* be used for this purpose. The RET key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. It will always provide a new line for you, however, and select a position at the left-hand side. The position where a letter or other character will appear when you press a key is indicated by the cursor.

You will find that the action of each key repeats if you hold your finger on it. You may find also that there is some 'keybounce', and that pressing a key will print a character more than once. On my MTX500 model, this often happened with the letter K. Practice tapping the keys, rather than holding them down, but if you find a lot of unwanted repetitions, then this is a fault and you will have to return the computer.

Cassette tryout

You can obtain a lot of enjoyment from a computer system that consists only of the machine and a TV receiver. Each time that you switch the machine off, however, all the program and other information that has been stored in the memory of the computer will be lost. Since it might take several hours to enter a program into the machine by typing instructions on the keyboard, this waste just has to be avoided. We avoid the loss of programs by recording them on cassette tape.

The computer has circuits which will convert the instructions of a program into musical tones, which can then be recorded on an ordinary cassette recorder. When these notes are replayed, another set of circuits will convert the signals back into the form of a program. In this way, the use of a cassette recorder allows you to record your program on tape and to replay it again. Before you tackle the rest of this book, then, it's important to check now that you can record and replay programs.

Almost any portable cassette recorder can be used, but if you are buying a recorder specially for computer use, get one which is mains or battery operated, has a tape counter and automatic recording level control. Most small portable cassette recorders have these features, but hi-fi and other stereo recorders are usually unsuitable. I have used a Curry's Trophy CR100 recorder for years with very satisfactory results. Boots sell a very similar machine under the code name of CR325. Recorders by Sanyo, Hitachi and Sharp have also proved satisfactory. If you are in any doubt, ask to see the recorder being used to save and load Memotech programs.

Start work by switching everything off. Now find the cassette lead of the Memotech. This has two small plugs at each end. These small plugs are colour-coded, one white, the other black. At the computer, the white plug goes into the socket that is marked EAR, and the

black plug into the socket that is marked MIC. These two sockets are labelled CASSETTE. At the cassette recorder, the white plug engages into the socket that is marked EAR, or which has a drawing of an ear. This is where the signals come out of the recorder to be sent to the computer. The black plug fits into the socket that is marked MIC, which is the microphone input of the cassette recorder. The Memotech uses only these two plugs. The colours aren't important, providing that you match them correctly at each end.

Once you have made this connection, the cassette recorder is ready for use. It's preferable to run the recorder from the mains because battery life can be unpredictable, and if your batteries decide to fail while you are recording, you will probably lose the program. The next thing that you have to sort out is a supply of blank cassettes. There's nothing wrong with using reputable brands of C90 length cassettes (ordinary ferric tape, not the hi-fi CrO_2 type), but you'll find that the short lengths of tape that are sold as C5, C10 or C15 in computer shops and in most branches of W. H. Smith's, Boots, and Currys are much more useful.

Put a fresh cassette into the machine, with the 1 or A side uppermost. The first part of the cassette tape consists of a 'leader', which is plain, not recording, tape. This has to be wound on before you can record. If your recorder has a tape counter, reset the counter to zero, and then fast-wind the cassette to a count of 5. If there is no tape counter, take the cassette out and insert the body of a BiC pen into the centre of the empty reel. Turn the pen so that the tape winds on to the reel, and keep turning until you see the brown recording tape replace the clear or brightly-coloured leader.

Now before you can make a recording to test the system, you need a program to record, and this involves some typing. This is easy if you have just switched the Memotech on, but if you have been pressing keys at random, then it's a good idea to switch off again, then on.

Type the number 1Ø (1 and then Ø), and then the word REM. Check that this looks correct, and then press the RET key. The effect of this is to place the instruction line 1Ø REM into the memory of the Memotech. As you type the first digit, the 'Ready' message will disappear, and the characters will be seen on the screen at the cursor position. When you press the RET key, the whole line shifts upwards on the screen. If you see a mistake as you type the line, just press the back-space key, which is marked BS and is on the top row of keys, right-hand side. This shifts the cursor backwards, and you can use it to place the cursor over an incorrectly typed letter. You can then

type the correct letter, which will replace the incorrect one. If this makes the line correct, pressing RET will enter it into the computer. If you get a 'Mistake' message, the cursor will return to the first letter of the REM command, and you can re-type it, correctly this time. The Memotech will not allow you to enter incorrect commands! Now type the rest of the lines, as illustrated in Fig. 1.6, remembering to press the RET key after you have completed typing each line. The numbers are called *line numbers*, and they are there for two reasons. One is to remind the computer that this is a program, the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers. An alternative method of obtaining the REM word is to press the key that is marked F1. By using the sequence of keys 1,Ø,F1,RET you will place the complete line of 1Ø REM into the memory.

Check that the program on the screen looks like the printed version in Fig. 1.6, and make sure that the cassette recorder is ready.

```
10 REM
20 REM
30 REM
40 REM
```

Fig. 1.6. A program for testing the cassette recording and replaying actions.

Now type SAVE"A". SAVE is the instruction to the computer meaning that you want to save (record) a program on a cassette. The "A" is a *filename* which the computer will use to recognise the program if it is asked to. You must put in the quotes (inverted commas, obtained by pressing SHIFT and 2 at the same time), otherwise the computer cannot carry out the instruction. You can, however, omit the A if you like, so that the command looks like SAVE"". Don't press RET yet! Now start the recorder by pressing its PLAY and RECORD keys. Press them firmly so that they lock in place, and you will see the reels of the cassette turning. A few machines, notably some SHARP models, only require you to press the RECORD key of the recorder, but this is unusual. When the recorder is working, press the RET key on the Memotech. If you have turned up the volume control of the TV receiver, you will hear a few bursts of sound. After a very short time, the cursor of the Memotech will reappear on the screen with a 'Ready' message underneath. This lets you know that the program has been recorded, and you can switch the recorder off. That's all. Now set the volume control of the recorder to three-quarters of the way up to its maximum range, and the tone control, if any, to maximum treble.

Now comes the crunch. You have to be sure that the recording was O.K. Wind back the tape again. Type NEW and press RET. This should have wiped your program from the memory. Now type LIST and press RET. Nothing should appear – LIST means put a list of the program instructions on the screen, and there shouldn't be any!

You can now load the instructions in from the tape. Type LOAD"" and press RET. Now press the PLAY key of the recorder (did you rewind the tape?). At the first burst of sound, you should see the message 'FOUND: A' appear on the screen, assuming that you used the filename of A when you recorded. This will be followed by the word 'LOADING' at the next burst of sound. If you don't see either of these messages, then it's likely that the volume control setting is too low. The 'Ready' message will reappear to show when the loading operation is complete. When this appears, the program is in place, and the recorder can be stopped. Type LIST now, then press the RET key. You should see your program appear on the screen. If all you see is a row of dots, or other rubbish, then the usual cause is that the volume control setting of the recorder was too low. The Memotech seems to need unusually high volume control settings.

The Memotech has also a VERIFY command, which is intended to check the quality of recordings. With the program in the memory (type LIST, then press RET to make sure), wind back the tape, using the rewind key of the recorder, and type:

VERIFY"A"

then press RET. This command will cause the Memotech to compare what is stored in its memory with the program that you recorded. It can't do so, however, until you play the program back. Press the PLAY key of the recorder, and wait. After a time, you should see the message 'VERIFYING' appear. When the screen shows the message 'Ready' again you can stop the recorder. Your cassette recording is O.K., and correct recordings are being made. If the program and the recording do not match perfectly, then you will see the message 'Mismatch'. If this happens, stop the tape, and press the CLS key, then RET. Adjust the volume control, and try again. I found that the VERIFY action was *very* fussy, and that recorded programs which loaded easily would still give 'Mismatch' messages with VERIFY. Once you have found a position for the volume control which gives no 'Mismatch' message, mark this setting and try to keep it unchanged. Remember that if you change your brand of tape, you may have to find a new volume control setting. I found

that once the correct settings were obtained, the cassette operation was unusually reliable.

Once you can reliably save programs on tape, verify them, and reload them, you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes more work will save your effort on tape so that you won't have to type it again.

What can go wrong? Bad connections, and incorrect volume control settings mainly, and the table in Fig. 1.7 should help you to trace the source of problems. Remember that very small changes of the volume control position can have considerable effects. There is one other problem, however, which you may find puzzling. Even though you can record and replay your own programs, you may find that when you buy a program on a cassette it refuses to load at any setting of the recorder's volume control. This is nearly always because you need your head looked at. That means the head of the cassette recorder, and there's advice on this problem in Appendix A.

Error check list

1. Are you sure that you have a program on the tape? Remove the EAR plug, and play the tape back. You should hear a musical tone which signals the start of the program, then a short burst of noise which is the program.
2. Have you tried several settings of the volume control? Some of the programs which I recorded would play back only at maximum volume control setting.
3. Check that you have the plugs the correct way round. They should be colour-coded, but the colour coding *might* possibly be wrong.
4. Check that your recorder is working properly by using the microphone to record a message, and then listening to the replay.
5. Always check by making a fresh recording and then verifying it. Be careful to separate your recordings from each other, because they take up only a small length of tape each.

Fig. 1.7. A checklist for recording faults.

Chapter Two
Screen Time

Chapter One will have broken you in to the idea that the Memotech, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the RET key is pressed. You will by now have used the command LIST which prints your program instructions on to the screen. There are two other useful points that you need to know before we go much further. One is that you can clear the screen by pressing the key marked F2 (one of the set of eight at the right-hand side), then RET. The other is the use of the key that is marked CLS, on the pad of number keys. The CLS key (bottom right on its set of 12) will allow you to escape from a faulty command. Suppose, for example, that you have typed LOADA, and pressed RET. The machine informs you that this is a mistake, and gives you a chance to correct it. You may, however, have decided that you want to do something different. By pressing the CLS key, then RET, you can get back to normal operation in an easier way than trying to erase a command word. This is necessary because the back-space key (marked BS) does not erase any letters on the screen. As your familiarity with the computer keyboard increases, you will want to make use of the editing commands, and these are explained in Appendix B.

Now, there are two ways in which you can use a computer. One way is called *direct mode*. Direct mode means that you type a command, press RET, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In program mode the computer is issued with a set of instructions and a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*. The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated

only if you type the whole command again, and then press RET. The set of command words that can be used, along with the rules for using them, make up what is called a *programming language.* The Memotech provides you with the most commonly-used of all programming languages for small computers, BASIC. BASIC is short for Beginners All-purpose Symbolic Instruction Code, and it was originally devised for teaching purposes. Since then, it has developed into a useful language in its own right. The version of BASIC that your Memotech uses is, however, enriched by a lot of extra commands. Some of these commands are similar to those used by another teaching language, called LOGO. In addition, the Memotech allows you to make use of another, more specialised, programming language. This is called NODDY, and is used for instructions and other pieces of text on the screen. There's a lot more about NODDY in Chapter Nine.

Let's now take a look at the difference between a direct command and a program instruction. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

PRINT 1.6 + 3.2 (and then press RET)

You have to start with PRINT, or its abbreviation P., because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like 'GIVE ME' or 'WHAT IS', only a few words that we call its *reserved words* or *instruction words.* PRINT is one of these words.

When you press RET after typing PRINT 1.6 + 3.2, the screen shows the answer, 4.8, under the command, and the word 'Ready' appears under this answer. The 'Ready' is a *prompt*, a reminder that the computer is ready for another command. Once this command has been carried out, however, it's finished.

A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press RET. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the difference between your commands and your program instructions. On computers that use the *language* called BASIC this is done by starting each program instruction with a number which is called a *line number.* This must be a positive whole number, the type of number that is called a positive integer. This is

why you can't expect the computer to understand an instruction like 5.6 + 3 = ; it takes the 5 as a line number, and the rest doesn't make sense.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results.

```
10 PRINT 5.6+6.8
20 PRINT 9.2-4.7
30 PRINT 3.3*3.9
40 PRINT 7.6/1.4
```

Fig. 2.1. A four-line arithmetic program.

Take a close look at this, because there's a lot to get used to in these four lines. To start with, the line numbers are 1Ø,2Ø,3Ø,4Ø rather than 1,2,3,4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 1Ø and line 2Ø, then you can type the line number 15 , or 11 or 12 or any other whole number between 1Ø and 2Ø, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 1Ø and 2Ø. If you number your lines 1, 2, 3 then there's no room for these second thoughts though you can change line numbers if you have to by using the editing commands.

The next thing to notice is how the number zero on the keyboard is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the Ø in place of O, nor the O in place of Ø, and the slashing makes this difference more obvious to you, so that you are less likely to make mistakes. The zero that you see on the screen *isn't* slashed, but it is six-sided, and the O is four-sided. Type some zeros and Os on the screen so that you can see the difference.

Now to more important points. The star or asterisk symbol in line 3Ø is the symbol that the Memotech uses as a multiply sign. Once again, we can't use the × that you might normally use for writing multiplication because this is a letter. There's no divide sign on the keyboard, so the Memotech, like all other small computers, uses the backslash (/) sign in its place. This is the diagonal line on the same key as the question mark, not the one on the key next to the BS key.

So far, so good. The program is entered by typing it, just as you

see it. You don't need to leave any space between the line number and the P of PRINT, because the Memotech will put one in for you when it displays the program on the screen. You *must,* however, leave a space following each command word. The Memotech will not accept a command unless you do this. If you have used any other computer previously, you may find this rather difficult to get used to. You will have to press the RET key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in Fig. 2.1. When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. There are two things that you need to know now. One is how to check that the program is actually in the memory, the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can use the F2, then RET, keys to wipe the screen first if you like, then type LIST and press the RET key. When you press the RET key, and not until, your program will be listed on the screen. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. To make the program operate, you need another command, RUN. Type RUN, then press the RET key, and you will see the instructions carried out. To be more precise, you will see:

```
12.4
4.5
12.87
5.42857143

Ready
```

When you follow the instruction word PRINT with a piece of arithmetic like 2.8 * 4.4 , then what is printed is the result of working out that piece of arithmetic. The program *doesn't* print 2.8 * 4.4, just the result of the action 2.8 * 4.4.

Now this is useful, but it's not always handy to get a set of answers on the screen, especially if you have forgotten what the questions were. The Memotech allows you a way of printing anything that you like on the screen, exactly as you type it, by the use of what is called a *string.*

Figure 2.2 illustrates this principle. In each line, some of the typing is enclosed between quotes (inverted commas) and some is

```
10 PRINT "2+2=";2+2
20 PRINT "2.5*3.5=";2.5*3.5
30 PRINT "9.4-2.2=";9.4-2.2
40 PRINT "27.6/2.2=";27.6/2.2
```

Fig. 2.2. Using quote marks. In this and other examples, the abbreviation P. was used in place of the PRINT instruction word, but PRINT appears in the listing.

not. Enter this short program and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

2 + 2 = 4

Now there's nothing automatic about this. If you type a new line:

15 PRINT "2+2=";5*1.5

then you'll get the daft reply, when you RUN this, of:

2 + 2 = 7.5

The computer does as it's told and that's what you told it to do. What looney thought that computers would take over the world?

This is a good point also to take notice of something else. The line 15 that you added has been fitted into place between lines 1Ø and 2Ø – LIST if you don't believe it. No matter in what order you type the lines of your program, the computer will sort them into order of ascending line number for you. Note also that the computer has put a space between the '=' and the number. This doesn't always happen, and this space is reserved for a minus sign. If we want to make sure that a number with a minus sign is separated from the '=' sign, we need to add a space between the '=' and the final quote-marks.

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, as far as the Memotech is concerned, always means print on to the TV screen. For activating a paper printer (hard copy, it's called), there's a separate instruction LPRINT (and LLIST for program listings). It's not an indication of Welsh design – the L once meant 'line' in the days when printers for computers were huge pieces of machinery that printed a whole line at a time. These instructions are not useful to you unless you have a printer connected.

Now try the program in Fig. 2.3. You can try typing the lines in

```
10 PRINT "THIS IS"
20 PRINT "THE REMARKABLE"
30 PRINT "MEMOTECH."
```

Fig. 2.3. Using the PRINT instruction to place words on the screen.

any order that you like, to establish the point that they will be in line-number order when you list the program. When you RUN the program, the words appear on three separate lines. This is because the instruction PRINT doesn't just mean print-on-the-screen. It also means 'take a new line', and start at the left-hand side!

Now this isn't always convenient, and we can change the action by using punctuation marks that we call *print modifiers*. Start this time by acquiring a new habit. Type NEW and then press the RET key. This clears the old program out, and it also clears the screen. If you don't do this, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. In Fig. 2.3, for example, the line 15 would be left in store even when you typed a new line 1Ø and a new line 2Ø.

Now try the program in Fig. 2.4. There's a very important difference between Fig. 2.4 and Fig. 2.3, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to

```
10 PRINT "THIS IS ";
20 PRINT "THE REMARKABLE ";
30 PRINT "MEMOTECH."
```

Fig. 2.4. The effect of semicolons.

prevent the next piece of printing starting on a new line at the left-hand side. When you RUN this program, all of the words appear in one line. It would have been a lot easier just to have one line of program that read:

1Ø PRINT "THIS IS THE REMARKABLE MEMOTECH."

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at examples of that sort of thing later.

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of Fig. 2.5 should not come as too much of a surprise. Line 1Ø contains a novelty, though,

```
10 CLS : PRINT "THIS IS MEMOTECH"
20 PRINT : PRINT
30 PRINT "_READY TO WORK FOR YOU."
```

Fig. 2.5. Clearing the screen with the CLS instruction, and using multistatement lines. The action of the CLS key is not the same as that of CLS in a program line.

in the form of two instructions in one line. The instructions are separated by a colon (:), and you can, if you like, have several instructions following one line number in this way, taking several screen lines. So long as the number of characters in the 'line' does not take up more than four lines on the screen, you can put instructions together in this way. In a *multistatement* line of this type, the Memotech will deal with the different instructions in a left-to-right order.

The other point about Fig. 2.5 is that line 2Ø causes the lines to be spaced apart. The two PRINT instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy. Remember, to save typing, that you can type P., and the machine will convert this into 'PRINT' for you!

Figure 2.6 deals with columns. Line 1Ø is a PRINT instruction that acts on the numbers 1,2,3,4 and 5. When these appear on the screen, though, they will be spaced out just as if the screen had been divided into five columns. The mark which causes this effect is the

```
10 PRINT 1,2,3,4,5
20 PRINT 1,2,3,4,5,6
30 PRINT "ONE","TWO","THREE","FOUR","FI
VE"
40 PRINT "THIS IS TOO LONG","TWO","THRE
E","FOUR","FIVE"
```

Fig. 2.6. How the comma causes words to be placed into five columns.

comma, and the action is completely automatic. As line 2Ø shows, you can't have six columns. Anything that you try to get into a sixth

column will actually appear on the first column of the next line down. The normal action works for words as well as for numbers, as line 3Ø illustrates. When words are being printed in this way, though, you have to remember that the commas must be placed outside the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into columns something that is too large to fit, the long phrases will spill over to the next column, and the next item to be printed will be at the start of the next column along. Line 4Ø illustrates this – the first phrase spills over from column 1 into column 2, and the word TWO is printed starting at column 3.

Commas are useful when we want a simple way of creating five columns. However, a much more flexible method of placing words on the screen exists. This makes use of the command word CSR (short for CurSoR) to position the cursor anywhere on the screen. For the purpose of using CSR, we need to remember that the screen is divided into a grid of 39 divisions across and up to 24 down. These are numbered as 0 to 38 across and 0 to 23 down. When you are entering programs, however, you can use only 19 lines down the screen, so you have to keep to the numbers 0 to 18 rather than 0 to 23. We'll see later how you can make use of more of the screen.

Figure 2.7 shows how CSR is used. It has to be followed by two numbers, separated by commas, and there must be a space between the R of CSR and the first number. The first number is the 'across'

```
10 CSR 15,2: PRINT "CENTRE"
20 CSR 2,10: PRINT "START HERE"
```

Fig. 2.7. How CSR is used to position the cursor.

number, or X number, 0 to 38. A value of Ø will place the cursor at the left-hand side, and a value of 38 will place it at the right-hand side. The second number is the 'down' number, or Y number, and in this illustration we will keep to numbers between 0 and 18. A value of Ø places the cursor on the top line, and a value of 18 places the cursor on the bottom line. Figure 2.8 illustrates the positions that you can get to with these numbers. If you don't feel like typing CSR each time you want to use it, you can get the command by pressing the SHIFT and F7 keys together. The word does *not* appear on the screen until the RET key has been pressed, and you have to remember to type the two numbers that follow CSR before you press RET. After you have used CSR to place the cursor where you

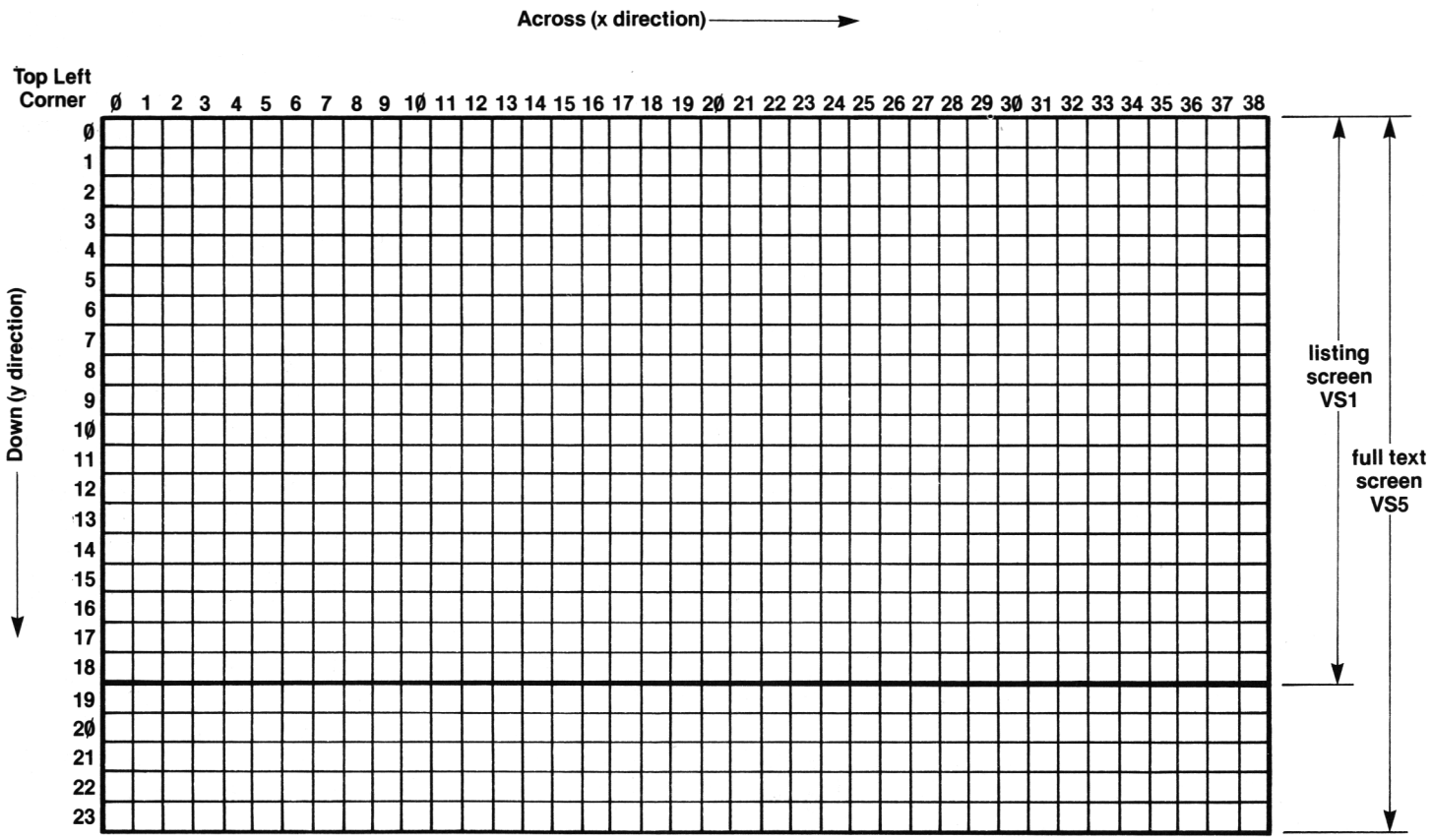


Fig. 2.8. The CSR map, showing how the CSR numbers are used.

want it, you can use PRINT (remember that P. is easier to type) to print a number or letter at the cursor position. You don't have to print in the order of left to right or top to bottom either, because CSR allows you complete freedom to place the cursor and print wherever you want. If your choice of CSR position places a new word over an old one, then the new letters will simply replace the old ones.

Oh, yes, how did I position the word 'CENTRE' at the centre of a line? This is done by calculating the correct 'across' number for the CSR instruction. The method is shown in Fig. 2.9. You have to

---

(1) Count number of characters in the title, including spaces.
(2) Subtract this number from 39.
(3) Divide the result by 2, ignoring any remainder. Now subtract 1.
(4) Use the result as the CSR number.

Example

Title: MEMOTECH DELIGHT! 17 characters.
Now 39 - 17 = 22, and half of 22 = 11. 11 - 1 = 10, so use CSR 10,5 to centre the title on line 5.

Fig. 2.9. The formula for centring a title.

count up the number of characters that you want to print centred. By characters, I mean letters, digits, spaces and punctuation marks. You then subtract this from 39, and divide the result by 2. Take the whole number part of the answer (forget about any half left over), and subtract 1 (because we number from ∅, not from 1). This is the first (across) number to use with CSR, and you can pick the second number so as to use the top line (∅), the next one down (1), or wherever else you like. In a later chapter, you will see that we can use letters in place of numbers in the CSR and other instructions. This allows us to centre words without all the fuss of counting letters – but that's more advanced programming than we should be thinking about at this point! For the moment, cast your eye and your typing fingers in the direction of Fig. 2.10, which shows how CSR can be used to place text anywhere on the screen and in any order.

At this point, I think I also owe you some explanation about the different numbers that you can use with CSR. The Memotech has its screen organised in a way that will be familiar to former users of the ZX-81 and Spectrum, but which may baffle newcomers, or former owners of other computers. The screen is divided, invisibly, into three sections when you are typing programs. One section of four

```
10 CLS
20 CSR 5,18: PRINT "FIRST ITEM"
30 CSR 2,9: PRINT "SECOND"
40 CSR 8,1: PRINT "THIRD"
```

Fig. 2.10. Illustrating how any order of printing can be used with CSR. You could even print words starting at the bottom of the screen and going upwards if you wanted to.

lines near the bottom of the screen is reserved for entry. As you type a command or program line, it appears on these four lines. Below this *entry screen*, there is another line, the *message screen*, which is used for the 'Ready', 'Mistake', and other messages. Above the *entry screen* is the main *list screen.* This consists, as we have seen, of 19 lines, and it's on these lines that you see programs listed, or immediate commands carried out. When you RUN a program, however, the whole screen is used momentarily. Whenever the program ends, though, the screen divisions return. You have to keep a close watch on the bottom line while you are programming, because this line will show you the messages that keep you in touch with what is happening.

Now try the program of Fig. 2.11 as an illustration of the use of lower-case letters. As we saw earlier, the Memotech will normally print upper-case letters if you type a letter whether you are holding down the SHIFT key or not. You can change this action by pressing the ALPHA LOCK key. When the ALPHA LOCK has been

```
10 CLS
20 CSR 16,2: PRINT "TITLE"
30 CSR 2,4: PRINT "This is an example o
f MEMOTECH"
40 PRINT "UPPER and lower case letters.
"
```

Fig. 2.11. Using lower-case. The ALPHA LOCK is a 'toggle' key, meaning that you press it once to get lower-case, and press it again to get back to upper-case. Any command word that you enter in lower-case is converted to upper-case when you press RET.

pressed once, letters will appear in lower-case unless you also press the SHIFT key. Press the ALPHA LOCK key again, and you will be back to the normal arrangement of upper-case letters at all times. You can make lower-case letters appear only when they are typed between quotes, and following a PRINT instruction. If you type instruction words in lower-case, you will see them changed to upper-case when the instruction is entered (when you press RET).

One last mystery. You will have noticed that the 3 key on the top row of the main keyboard has a pound sign on it. You do *not* see a pound sign when you press the SHIFT key and the 3 key, however. What you see is the *hash* sign, #. That's because the Memotech needs the hash sign for some types of instructions (not listed in this book). The hash sign is, in fact, part of the standard international (which means American!) character set. The Memotech, unique among small computers, offers you a chance to change this if you want. If you do not use the hash sign in programs, you can get a UK set of characters by pressing ESC, then B, then 1. Do not hold the keys down together, just press them in sequence. This sequence is written as ESC B1. After doing this, you will find that the SHIFT 3 action gives you the pound sign. Several other different character sets can be obtained in this way, and they are listed in the manual. When you use the other sets (numbers 2 to 5 following the B), you will find that the differences are in the symbols on the^ key, and on the [and] keys, with or without SHIFT. You can return to the international set by using ESC Bø. Do *not* use ESC S as the provisional copy of the manual suggests, because this locks up the keyboard, and you have to press the two reset keys to get back to normal! If you have a program in the memory at the time, it will be lost because of this reset action, so be careful when you make such experiments. Always save a program before trying out anything new.

Chapter Three

A Bit of Variation

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that go on before anything is printed. One of these is called *assignment*. Take a look at the program in Fig. 3.1. Type it in, RUN it, and contrast what you

```
10 CLS
20 LET X=23
30 PRINT "2 TIMES";X;" IS";2*X
40 LET X=5
50 PRINT "X IS NOW";X
60 PRINT "AND TWICE";X;" IS";2*X
```

Fig. 3.1. Assignment in action. The letter X has been used in place of a number.

see on the screen with what appears in the program. The first line that is printed is line 3Ø. What appears on the screen is:

2 TIMES 23 IS 46

but the numbers 23 and 46 don't appear in line 3Ø! This is because of the way we have used the letter X as a kind of code for the number 23. The official name for this type of code is a *variable name.*

Line 2Ø assigns the variable name X, giving it the value of 23. *Assigns* means that wherever we use X, not enclosed by quotes, the computer will operate with the number 23. Since X is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X = 2174.3256, for example. Line 3Ø then proves that X is taken to be 23, because wherever X appears, not between quotes, 23 is printed, and the *expression* 2*X is printed as 46. We're not stuck with X as representing 23 for ever, though. Line 4Ø assigns X as being 5, and lines 5Ø and 6Ø prove that this change has been made.

That's why we call X a *variable* – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. Even after you have run the program of Fig. 3.1, providing you haven't added new lines or deleted any part of it, you can type PRINT X (or PRINTX), and pressing RET will show the value of X on the screen.

This very useful way to handle numbers in code form can use a *name* which must start with a letter. You can add to that letter other letters or digits, but not spaces or punctuation marks, so that N, NAme, N53TOORISKY are all names that you can use for number variables, and each can be assigned to a different number. Just to make it even more useful, you can use similar 'names' to represent words and phrases also. The difference is that you have to add a dollar sign ($) to the variable name. If N is a variable name for a number, then N$ (pronounced en-string or en-dollar) is a variable name for a word or phrase. The computer treats these two, N and N$, as being entirely separate and different.

Each assignment of value to a name *must* be done by using LET, and there must be a space between the 'T' of LET and the first letter of the name. If you are a beginner, you soon get used to this, but if you have used other computers, apart from the ZX-81 or Spectrum, you may find it takes you a long time! Fortunately, the Memotech reminds you that LET is needed, and will not allow you to enter a line with the LET missing.

Serenade for strings

Figure 3.2 illustrates *string variables*, meaning the use of variable names for words and phrases. Lines 1Ø and 2Ø carry out the

```
10 CLS : LET N$="Memotech"
20 LET A$="MTX500": LET B$="MTX512"
30 CSR 9,2: PRINT "THE ";N$;" FAMILY."
40 PRINT : PRINT ,A$,,B$
50 PRINT : PRINT "The new wave in compu
ters."
```

Fig. 3.2. Using string variables. These are distinguished by the dollar sign.

assignment operations, and lines 3Ø to 5Ø show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which *must* be surrounded by quotes.

Figure 3.3 shows another example, this time using the variable names MEMOTECH$ and OPINION$ for longer phrases. There wouldn't be much point in printing messages in this way if you wanted the message once only, but when you continually use a

```
10 CLS : LET MEMOTECH$="The new compute
r"
20 LET OPINION$="is a real step forward
."
30 PRINT "MEMOTECH, ";MEMOTECH$;"   "
40 PRINT OPINION$
```

Fig. 3.3. Illustrating variable names with more than two letters. These can be very useful to remind you of what the name means.

phrase in a program, this is one method of programming it so that you don't have to keep typing it! One of the great joys of the Memotech is that it is one of the few computers that allows you to use real 'names' for your variables, rather than just single or double letters, as Fig. 3.3 illustrates. By doing this, you can make life a lot easier for yourself by using names which have a meaning. It's easier, for example, to understand what's happening in a program when a variable is called TOTALINPUT than if it is called just A. There's nothing to stop you from using variable names of as many characters as you like. Nothing, except the fact that it uses up precious memory, and that you have to stick to letters and numbers. You cannot use punctuation marks to make names easier to read. You can't, for example, use FINAL-TOTAL, because the Memotech will not accept the dash as part of a variable name.

Strings and things

Because the name of a string variable is marked by the use of the $ sign, a variable like A$ is not confused with a number variable like A. We can, in fact, use both on the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the difference is a bit more than skin deep, though. Lines 1Ø and 2Ø assign number variables A and B, and string variables A$ and B$. When these variables are printed, you can't tell the difference between A and A$ or between B and B$. The difference appears, however, when the computer attempts to complete line 6Ø. It can multiply two number variables, because numbers can be multiplied, but it can't multiply string variables. It will not, in fact, even accept a

```
10 LET A=2: LET B=3
20 LET A$="2": LET B$="3"
30 CLS
40 PRINT A;" TIMES";B;" IS";A*B
50 PRINT
60 PRINT A$;" TIMES ";B$;" IS NOT ACCEP
TED!"
```

Fig. 3.4. String and number variables might look alike when they are printed, but they are different!

line that contains such an instruction! The reason is simple. A string variable can be anything. We have assigned A$ as '2', but we could just as easily have assigned it as '2 LABURNUM WAY'. You can multiply 2 by 3, but you can't multiply 2 LABURNUM WAY by 3 ACACIA AVENUE. The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings. Attempting to do a forbidden operation in line 60 causes an error message when you press RET to enter the line. The word 'Mistake' appears on the bottom line of the screen, and the cursor flashes over the asterisk as a reminder to you of where the mistake is. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause an error message. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's only a machine!

There is one operation, that looks rather like arithmetic being carried out on strings. It uses the + sign, but it isn't addition in the sense of adding numbers. Figure 3.5 illustrates this action of joining strings, which is often called *concatenation.* This is nothing like the action of arithmetic, as you'll see if you use numbers in place of the names placed between the quotes. Concatenation is a very useful

```
10 LET A$="SMITH"
20 LET B$="JONES"
30 CLS
40 PRINT : PRINT "JUST CALL ME ";A$+"-"
+B$;", HE SAID."
```

Fig. 3.5. Concatenating or joining strings. This is not the same action as addition!

way of obtaining strings which otherwise would need rather a lot of typing. Take a look at Fig. 3.6. This defines strings A$ and B$ as characters which can be used as 'frames' around a title. Remember that the hash mark, #, is obtained by using SHIFT 3, it's shown as a

```
10 LET A$="***": LET B$="###"
20 LET M$="MEMOTECH"
30 CLS
40 CSR 9,3: PRINT A$+B$+M$+B$+A$
```

Fig. 3.6. Using concatenation to make a frame for a title.

pound sign on the keyboard. The title is defined in line 2Ø as MEMOTECH. Line 4Ø then prints a concatenated string. This has needed less typing than if you had to type all the characters between the quotes. It also allows you to rearrange the frames as you please. You can, for example, use:

B$ + A$ + M$ + A$ + B$

next time you print the title. Notice how I have used CSR 9,3 to ensure that the word appeared centred on the line.

Getting some in

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

Figure 3.7 illustrates this with a program that prints your name. Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

WHAT IS YOUR NAME

are printed on the screen. On the line below this you will see a question mark. The computer is now waiting for you to type something, and then press RET. Until the RET key is pressed, the program will hang up at line 3Ø, waiting for you. If you're honest, you will type your own name and then press RET. You don't have to put quotes around your name, simply type it in the form that you

```
10 CLS
20 PRINT "WHAT IS YOUR NAME"
30 INPUT NAME$
40 CLS : PRINT : PRINT
50 PRINT NAME$;" -THIS IS YOUR LIFE!!"
```

Fig. 3.7. Using the INPUT instruction. The name that you type is put into the phrase in line 5Ø.

want to see printed. When you press RET, your name is assigned to the variable NAME$. The program can then continue, so that line 4Ø clears the screen and spaces down by two lines. Line 5Ø then prints the famous phrase with your name at the start.

You could, of course, have answered MICKEY MOUSE or DONALD DUCK or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. A computer is a type of robot, a machine that does as it has been instructed. When you get a gas bill for a million pounds, or the wrong name on a car tax form, it's not because some computer was being silly!

We aren't confined to using string variables along with INPUT. Figure 3.8 illustrates an INPUT step which uses a number variable N. The same procedure is used. When the program hangs up with the

```
10 PRINT "ENTER A NUMBER"
20 INPUT N
30 PRINT
40 PRINT "TWICE";N;" IS";2*N
```

Fig. 3.8. An INPUT to a number variable. The quantity that you type must be a number.

question mark appearing, you can type a number and then press the RET key. The action of pressing RET will assign your number to N, and allow the program to continue. Line 4Ø then proves that the program is dealing with the number that you entered. When you use a number variable in an INPUT step, then what you have typed when you press RET must be a number. If you attempt to enter a string, the computer will refuse to accept it. Some computers stop running at this point, but the Memotech simply prints another question mark on the next line down, and this gives you another chance to type a number and press RET again. If your INPUT step uses a string variable then anything that you type will be accepted when you press RET.

The way in which INPUT can be placed in programs can be used

to make it look as if the computer is paying some attention to what you type. Figure 3.9 shows an example – but with INPUT used in a different way. This time, there is a phrase following the INPUT instruction. The phrase is placed between quotes, and is followed by

```
10 CLS
20 INPUT "TYPE YOUR NAME, PLEASE ";NAME
$
30 PRINT
40 PRINT "VERY PLEASED TO MEET YOU, ";N
AME$
```

Fig. 3.9. Using INPUT to print a phrase which requests the input.

a semicolon and then the variable name NAME$. This line 2Ø has almost the same effect as the two lines:

```
15 PRINT "TYPE YOUR NAME, PLEASE";
2Ø INPUT NAME$
```

The difference is that no question mark is printed when you use INPUT with a phrase in this way. This allows you a lot more freedom, because you may not want to have a question mark. In this example, for instance, the appearance of a question mark would make the phrase look rather peculiar. If you want a question mark, then you can incorporate it into the phrase, as, for example,

```
5Ø INPUT "Your name?";NAME$
```

The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.10, for example, shows two variables being used after one INPUT. One of the variables is a

```
10 CLS
20 INPUT "NAME AND NUMBER, PLEASE ";NAM
E$,NUMBER
30 PRINT : PRINT
40 PRINT "THE NAME IS ";NAME$
50 PRINT : PRINT "THE NUMBER IS ";NUMBE
R
```

Fig. 3.10. Putting in two variables in one INPUT step.

string variable NAME$, the other is the number variable NUMBER. Now when the computer comes to line 2Ø, it will print the message and then wait for you to enter both of these quantities, a

name and then a number. There is only one way of entering these quantities. You must type the name, then a comma, and then the number. Pressing the RET key will then assign the two variables, and the computer will continue on its way. If you get it wrong, as for example by typing only the name and then pressing RET, you get another chance in the form of a question mark. If, incidentally, you want to break out of this, because you don't want to use the INPUT step, then you can press the BRK key. In the example of Fig. 3.10, when you have entered the name and number correctly, the name and number will be printed again in lines 4ϕ and 5ϕ.

We can extend this principle further. Figure 3.11 calls for four numbers to be entered. These, once again, must be entered by typing

```
10 CLS
20 INPUT "FOUR NUMBERS, PLEASE ";A,B,C,
D
30 PRINT
40 PRINT "THE SUM OF THESE IS ";A+B+C+D
```

Fig. 3.11. An INPUT step which calls for four numbers. These must be entered in one operation.

each number, then a comma, then the next number and pressing RET only after typing the last number. When you press RET, the program will print the sum in line 4ϕ.

Reading the data

The INPUT instruction is the one that we use for feeding information into the program from the keyboard while the program is running. There's yet another way of getting data into a program while it is running, though. This one involves reading items from a list that is included in the program, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. This word can be obtained by using SHIFT F8 if your typing hand is tired. The items of the list can be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the next item on the list.

We'll look at this in more detail in Chapter Five, but for the moment we can introduce ourselves to the READ...DATA instructions. Figure 3.12 uses the instructions in a very simple way.

```
10 CLS
20 READ NAME$
30 PRINT NAME$;
40 PRINT " IS VALUED AT ";
50 READ PRICE
60 PRINT PRICE;
70 PRINT " POUNDS."
80 DATA  GOLD WATCH, 650
```

Fig. 3.12. Using the READ and DATA words to place information into a program.

Line 2Ø reads the first item on the list and assigns it to the variable NAME$. This is printed in line 3Ø, with the semicolon keeping printing in the same line so that the phrase in line 4Ø follows it. The semicolon at the end of line 4Ø once more keeps the printing in the same line, and line 5Ø reads the number which is the second item in the list. This is assigned to the variable name PRICE (we could just as easily have used NAME$ again) and printed in line 6Ø. Once again, a semicolon prevents a fresh line from being taken, so that the final word of line 7Ø is printed following the number.

The READ...DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now. As before, though, we have to match the data items with the variable names that we use for them. We can read a number item and assign it to a string variable name, but we can't read a string item and assign it to a number variable name. If you try to do this, the program stops, and the message line of the screen shows the word 'Mismatch'. At the same time, the entry screen shows the line which has the READ instruction. Now, you have to be careful here, because this line may be correct, and the mistake may be in the DATA line. The computer can't be expected to know which line is incorrect, only that they don't match. If you want to alter the name of the variable in the READ line, well and good. If you want to alter the DATA line, then press RET to discard the READ line, and use an EDIT command to correct the DATA line. Editing is dealt with in Appendix B.

Number antics

The amount of computing that we have done so far should have

persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for word processing or even for accounts. It's time, then, to take a very brief look at the number abilities of the Memotech. It is a brief look because we simply don't have space to explain what all the mathematical operations do. In general if you understand what a mathematical term like Sin or Tan or Exp means, then you will have no problems about using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of *incrementing* if you are counting up and *decrementing* if you are counting down. Incrementing a number means adding 1 to it, decrementing means subtracting 1 from it. These actions are programmed in a rather confusing-looking way in BASIC, as Fig. 3.13 shows. Line 2Ø sets the value of X as 5. This is

```
10 CLS
20 LET X=5
30 PRINT "THE VALUE OF X IS ";X
40 LET X=X-1: PRINT
50 PRINT "AFTER USING X=X-1 ";
60 PRINT "THE VALUE OF X IS ";X
```

Fig. 3.13. Decrementing, using the equals sign to mean 'becomes'.

printed in line 3Ø, but then line 4Ø *decrements* X. This is done using the odd-looking instruction: LET X = X − 1, meaning that the new value that is assigned to X is 1 less than its previous value. The rest of the program proves that this action of decrementing the value of X has been carried out.

The use of the = sign to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

LET X = X+1

and this would have the effect of making the new value of X one more than the old value. X has been incremented this time. We could also use LET X = 2*X to produce a new value of X equal to double the old value, or LET X = X/3 to produce a new value of X equal to

the old value divided by three. Figure 3.14 shows another assignment of this type, in which both a multiplication and an addition are used to change the value of X.

```
10 CLS
20 LET X=5: PRINT "X IS ";X
30 PRINT
40 LET X=2*X+.5
50 PRINT "IT HAS CHANGED-": PRINT
60 PRINT "X IS NOW ";X
```

Fig. 3.14. A more elaborate reassignment, using an *expression*.

Number functions

Figure 3.15 illustrates some number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 1Ø picks the value of 2.5 for X. Line 2Ø then

```
10 CLS : LET X=2.5
20 PRINT "X SQUARED IS ";X^2
30 PRINT
40 PRINT "THE SQUARE ROOT IS ";SQR(X)
50 PRINT
60 PRINT "THE LOG. OF X IS ";LN(X)/LN(1
0)
```

Fig. 3.15. Using some number functions.

prints the value of X squared, meaning X multiplied by X. This is programmed by typing X^2, using the 'upside-down V' shape which is placed two keys to the right of the zero on the top line of keys. To get the square root of the number that has been assigned to X, we use the instruction word SQR, with the number in brackets. An alternative is X^.5, but SQR(X) is easier to type and remember. For other roots, like the cube root you can use expressions like X^(1/3) and so on. LN(X) produces the natural logarithm of X, and LN(X)/LN(1Ø) gives the more familiar logarithm to base 10.

Figure 3.16 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of interest only if you are interested in programming for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs.

Before we leave the use of numbers, though, there's another

ABS (X): Converts negative sign to positive.
ATN (X): Gives angle (in radians) whose tangent is X.
COS (X): Gives the cosine of angle X (radians).
EXP (X): Gives the value of e to the power X.
INT (X): Gives the whole-number part of X.
LN (X): Gives the natural logarithm of X.
MOD (X, Y): Gives the remainder after X has been divided by Y.
PI: Gives a value for the number pi.
RAND (X): Calculates fractions at random, using the number X as a basis for calculation. The sequence of fractions will eventually repeat.
RND: Gives a random fraction between ∅ and 1.
SGN (X): Gives the sign of X. The result is +1 if X is positive, -1 if X is negative, ∅ if X is zero.
SIN (X): Gives the sine of angle X (radians).
SQR (X): Gives the square root of X.
TAN (X): Gives the value of the tangent of angle X (radians).

Fig. 3.16. Memotech number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

important point, which concerns the order in which things are done. If you use an instruction such as:

```
P. 2*3 + 5/2
```

then what you get when you press RET is the value 8.5. The multiplication and division have been done first, then the addition. You will not get the answer of 8, which is what you get by adding 3 and 5, then multiplying by two and dividing by two. The computer, like a calculator, is designed so as to carry out arithmetic operations

For ordinary arithmetic, the order of priority is MDAS - multiplication and division, followed by addition and subtraction. The full order of priority is:

(1) Raising to a power, using ^ .
(2) Multiplication and division.
(3) Addition and subtraction.
(4) Comparison, using = <>
(5) AND
(6) OR
(7) NOT

Fig. 3.17. The order of priority for number actions. You can over-rule this order by placing any action within brackets.

in priority order, and this order is summarised in Fig. 3.17. You can break out of this order by using brackets. Whatever you enclose in brackets will be worked out before anything else is done. For example, if you type:

```
P. 2*(3 + 5)/4
```

and then press RET, you will get the answer of 4. The result of 3 + 5 = 8 has been worked out first, then the 2* and the /4 to get the answer. If you have one set of brackets inside another, then whatever is inside the innermost set is worked out first, then what is in the outer brackets, and so on in order.

How precise?

One of the problems of small computers is precision of numbers. You probably know that the fraction 1/3 cannot be expressed exactly as a decimal. How near we can get to its true value depends on the number of decimal places we are prepared to print, so that 0.33 is closer than 0.3, and 0.333 is closer still. The computer converts most of the numbers it works with into the form of a fraction and a multiplier. The fraction is not a decimal fraction but a special form called a binary fraction, and this conversion is seldom exact. The conversion is particularly awkward for numbers like 1, 10, 100 and also .1, .01, .001; all the powers of ten, in fact. To avoid embarrassments like printing 3–2=.9999999, the computer will round numbers of this type up or down as need be before using or displaying them. Not all computers do this well – you can be glad that you bought a Memotech!

Chapter Four
Repetitions and Decisions

Loops

One of the activities for which a computer is particularly well suited is repeating a set of instructions. Every computer is therefore well equipped with instructions that will cause repetition, and the Memotech is no exception. We'll start with the simplest of these *repeater* actions, GOTO.

GOTO means exactly what you would expect it to mean – go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again.

Figure 4.1 shows an example of a very simple repetition or *loop,* as well call it. Line 1Ø contains a simple PRINT instruction. When

```
10 PRINT "MEMOTECH MEMOTECH MEMOTECH ME
MOTECH"
20 GOTO 10
30 REM PRESS BRK TO STOP
```

Fig. 4.1. A very simple loop. You can stop this by pressing the BRK key.

line 1Ø has been carried out, the program moves on to line 2Ø, which instructs it to go back to line 1Ø again. This is a never-ending loop, and it will cause the screen to fill with the word MEMOTECH until you press the BRK key to 'break the loop'. Any loop that appears to be running forever can normally be stopped by pressing the BRK key, though if this does not work, you will have to press the two RST keys together, and lose the program. Note that when you use GOTO, there must be a space between GOTO and the line number that you

want to go to. I stress this point, because most computers allow instructions like GOTO1Ø, but the Memotech insists on GOTO 1Ø.

Now try a loop in which there is slightly more noticeable activity. Figure 4.2 shows a loop in which a different number is printed out each time the computer goes through the actions of the loop. We call

```
10 CLS : LET N=10
20 PRINT N
30 LET N=N-1
40 GOTO 20
50 REM BRK NEEDED AGAIN
```

Fig. 4.2. A loop which carries out a countdown action very rapidly. You will also have to use the BRK key to stop this one.

this *each pass through the loop*. Line 1Ø sets the value of the variable N at 1Ø. This is printed in line 2Ø, and then line 3Ø decrements the value of N. Line 4Ø forms the loop, so that the program will cause a very rapid countdown to appear on the screen. Once again, you'll have to use the BReaK key to stop it.

Now, an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we prefer not to use! We don't always have an alternative, but there is one – the FOR...NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The instructions that are repeated are the instructions that are placed between FOR and NEXT. Figure 4.3 illustrates a very simple example of the FOR...NEXT loop in action. The line which contains FOR must

```
10 CLS
20 FOR N=1 TO 10
30 PRINT "MEMOTECHNOLOGY"
40 NEXT
```

Fig. 4.3. Using the FOR ... NEXT loop for a counted number of repetitions.

also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, N is the counter variable, and its limit numbers are 1 and 1Ø. The NEXT is in line 4Ø, and so anything between lines 2Ø and 4Ø will be repeated.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print MEMOTECHNOLOGY ten times. At the first pass through the loop, the value of N is set to 1, and the phrase is printed. When the

NEXT instruction is encountered, the computer increments the value of N, from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 1Ø that has been set. If it doesn't, then line 3Ø is repeated, and this will continue until the value of N exceeds 1Ø – we'll look at that point later. The effect in this example is to cause ten repetitions.

You don't have to confine this action to single loops either. Figure 4.4 shows an example of what we call *nested loops*, meaning that one loop is contained completely inside another one. When loops are

```
10 CLS
20 FOR N=1 TO 10
30 PRINT "COUNT IS ";N
40 FOR J=1 TO 1000: NEXT
50 CLS : NEXT
```

Fig. 4.4. A program that uses nested loops, with one loop inside another.

nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 2Ø, using variable N which goes from 1 to 1Ø in value. Line 3Ø is part of this outer loop, printing the value that the counter variable N has reached. Line 4Ø, however, is another loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J, and we have put nothing between the FOR part and the NEXT part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The last action of the main loop is clearing the screen in line 5Ø. The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. In this example we have used NEXT to indicate the end of each loop. We could use NEXT J in line 4Ø and NEXT N in line 5Ø if we liked, but this is not essential. It also has the effect of slowing the computer down, though the effect is not important in this program. When you do use NEXT J and NEXT N, you must be absolutely sure that you have put the correct variable names following each NEXT. If you don't, the computer will stop with a 'No For' error – meaning that it can't match the NEXT with its FOR.

Even at this stage it's possible to see how useful this FOR...NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction

word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

FOR N = 1 TO 9 STEP 2

which would cause the values of N to change in the sequence 1, 3, 5, 7, 9. When we don't type STEP, the loop will always use increments of 1.

Figure 4.5 illustrates an outer loop which has a step of −1, so that the count is downwards. N starts with a value of 10, and is

```
10 CLS
20 FOR N=10 TO 0 STEP -1
30 PRINT N;" seconds and counting"
40 FOR J=1 TO 1000: NEXT
50 CLS : NEXT
60 PRINT "BLASTOFF!!"
```

Fig. 4.5. A countdown program, making use of STEP.

decremented on each pass though the loop. Line 40 once again forms a time delay so that the countdown takes place at a civilised speed. This is a particularly useful way of slowing the count down. A loop that uses a count of 1000 causes a delay of roughly one second. This isn't exactly constant, but it's good enough for a lot of operations that don't depend on very exact timing.

Every now and again, when we are using loops, we find that we need to use the value of N after the loop has finished. It's important to know what this will be, however, and Fig. 4.6 brings it home. This contains two loops, one counting up, the other counting down. At the end of each loop, the value of the counter variable is printed. This reveals that the value of N is 6 in line 50, after completing the FOR N = 1 TO 5 loop, and is 1 in line 90 after completing the FOR N = 5 TO 1 STEP −1 loop. A count-up loop, then, uses all the values

```
10 CLS
20 FOR N=1 TO 5
30 PRINT N
40 NEXT
50 PRINT "N IS NOW ";N
60 FOR N=5 TO 1 STEP -1
70 PRINT N
80 NEXT
90 PRINT "N IS NOW ";N
```

Fig. 4.6. Finding the value of the loop variable after a loop action is completed.

of N that are specified in the FOR...TO part of the instruction. A countdown, however, does *not* include the last number! The count from 5 to 1 prints 2 as its last figure. This is something that only the Memotech does, so that if you have used any other computer before, you will have to be careful. If you want to make use of the value of N, or whatever variable name you have selected to use, you will have to remember that it will have changed by one more step at the end of a count-up loop, but it will have reached the end number of a count-down loop.

One of the most valuable features of the FOR...NEXT loop, however, is the way in which it can be used with number variables instead of just numbers. Figure 4.7 illustrates this in a simple way.

```
10 CLS
20 LET A=2: LET B=5: LET C=10
30 FOR N=A TO B STEP B/C
40 PRINT N
50 NEXT
```

Fig. 4.7. A loop instruction that is formed with number variables.

The letters A, B and C are assigned as numbers in the usual way in line 2Ø, but they are then used in a FOR...NEXT loop in line 3Ø. The limits are set by A and B, and the step is obtained from an expression, B/C. The rule is that if you have anything that represents a number or can be worked out to give a number, then you can use it in a loop like this. You can, in fact, use variables to replace numbers in all of the instructions that call for numbers to be used.

Loops and decisions

It's time to see loops being used rather than just demonstrated. A simple application is in totalling numbers. The action that we want is to be able to enter numbers and have the computer keep a running total, adding each number to the total of the numbers so far. From what we have covered so far, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set. The program of Fig. 4.8 does just this.

The program starts by setting a number variable TOTAL to zero. This is the number variable that will be used to hold the total, and it

```
10 LET TOTAL=0: CLS
20 CSR 5,2: PRINT "Totalling numbers pr
ogram."
30 PRINT : PRINT "Enter each number as
requested"
40 PRINT "The program will show the tot
al."
50 FOR N=1 TO 10
60 PRINT "Number ";N;" please ";
70 INPUT J: LET TOTAL=TOTAL+J
80 NEXT
90 PRINT : PRINT "Total is ";TOTAL
```

Fig. 4.8. A number totalling program for ten numbers.

has to start at zero. As it happens, the Memotech arranges this automatically at the start of a program, but it's a good habit to ensure that everything that has to start with some value actually does. Lines 2Ø to 4Ø then issue instructions, and the action starts in line 5Ø. This is the start of a FOR...NEXT loop which will repeat the actions of lines 6Ø and 7Ø ten times. Line 6Ø reminds you of how many numbers you have entered by printing the value of N each time, and line 7Ø allows you to INPUT a number which is then assigned to variable name J. This is then added to the total in the second half of line 7Ø, and the loop then repeats. At the end of the program, the variable TOTAL contains the value of the total, the sum of all the numbers that have been entered.

It's all good stuff, but how many times would you want to have just ten numbers? It would be a lot more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like Ø or 999. A value like this is called a *terminator*, something that is obviously not one of the normal entries that we would use, but just a signal.For a number-totalling program, a terminator of Ø is very convenient, because if it gets added to the total it won't make any difference.

Figure 4.9, therefore, shows an example of this type of program in action. We can't use a FOR...NEXT loop, because we don't know in advance how many times we might want to go through the loop, so we have to go back to using GOTO. This time, however, we'll keep GOTO under closer control by putting a condition on it. This time the instructions appear first, but we still have to make the total variable TOTAL equal to zero in line 4Ø. Each time you type a number, then, in response to the request in line 5Ø, the number that you type is added to the total in line 6Ø, and line 7Ø prints the value

```
10 CLS : CSR 8,1: PRINT "Another Total
Finder"
20 CSR 2,3: PRINT "The program will tot
al numbers for you"
30 PRINT "until you enter a zero."
40 LET TOTAL=0
50 INPUT "Number, please ";NR
60 LET TOTAL=TOTAL+NR
70 PRINT "Total so far is ";TOTAL
80 IF NR<>0 THEN  GOTO 50
```

Fig. 4.9. A running-total program which can't use FOR ... NEXT.

of the total so far. Line 8Ø is the loop controller, and the key to the control is the instructions word IF. IF is used to make a test, and the test in line 8Ø is to see if the value of NR is not equal to zero. The odd-looking sign that is made by combining the *less than* and the *greater than* signs, <>, is used to mean *not equal*, so the line reads: 'if NR is not equal to zero, then goto line 5Ø'. The GOTO has to be in place, you can't just type 'THEN 5Ø' as you would with some other computers.

The effect, then, is that if the number which you have typed in line 5Ø was not a zero, line 8Ø will send the program back to repeat line 5Ø. This will continue until you do enter a zero. When this happens, the test in line 8Ø fails (NR is zero), and the program looks for a line 9Ø. Since it can't find one, it stops. This kind of action is sometimes called a *repeat...until* loop.

Now, this allows you much more freedom than a FOR...NEXT loop, because you are not confined to a fixed number of repetitions. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now.

Decisions, decisions

We can make a number of types of comparisons between number variables or numbers, and these are listed in Fig. 4.10. The mathematical signs are used for convenience, and you have to remember which way round the greater than and less than signs have to be. It's important to note that the equals sign means 'identical to' when it is used in a test like this. If A is 3.9999999 and B is 4.ØØØØØØØ then a test such as IF A = B will fail – A is not identical to B, even though it is close enough to be equal to our eyes. The important point here is that the numbers we see on the screen have

Sign	*Meaning*
=	Exact equality.
>	Left-hand quantity greater than right-hand quantity.
<	Left-hand quantity less than right-hand quantity.
	The signs can be combined as follows:
<>	Quantities not equal.
>=	Left-hand side greater than or equal to right-hand side.
<=	Left-hand side less than or equal to right-hand side.

Note: When the < or > sign is combined with =, then the < or > sign *must* be used first. Using a combination like => will always cause an error message.

Fig. 4.10. The mathematical signs that are used for comparing numbers and number variables.

been rounded, so that PRINT A in the example above might give the result 4. The test, however, is made on the numbers which have not been rounded.

Figure 4.11 shows another test – this time on string variables. The

```
10 CLS
20 PRINT "Press the Y or N key _ "
30 PRINT "then RET"
40 INPUT A$
50 IF A$="Y" THEN  PRINT "That's YES"
60 IF A$="N" THEN  PRINT "That's NO"
```

Fig. 4.11. Testing string variables, in this example to find whether a reply is Y or N.

instructions are in lines 2Ø to 3Ø, you are asked to type the Y or N key. Line 4Ø gets your answer; you have to type Y or N and then press RET. The key that you have pressed has its value assigned to A$, so that A$ should be Y or N. Lines 5Ø and 6Ø then analyse this result. If the key that you pressed was neither Y nor N, nothing is printed by the line 5Ø or 6Ø.

The test in this example is for identity. Only if A$ is absolutely identical to Y will the phrase 'That's YES' be printed. If you typed a space ahead of Y, or a space following, or typed y in place of Y, then A$ will not be identical, and the test fails. Failing means that A$ is not identical to Y and everything that follows THEN in that line will

be ignored. It's up to you to form these tests so that they behave in the way that you want!

The Memotech is one of a select group of computers that allows you to extend this IF...THEN test. The extension consists of the instruction word ELSE, and it offers an alternative to the test that is carried out by IF. Figure 4.12 illustrates this in action, with another Y/N program. The key line here is line 4Ø, where we have a pair of

```
10 CLS
20 PRINT "Type Y or N, then RET"
30 INPUT A$
40 IF A$="Y" THEN  GOTO 100 ELSE  IF A$
="N" THEN  GOTO 200
50 PRINT "Your answer of ";A$;" is not
Y or N -": PRINT "-please try again.":
GOTO 30
60 STOP
100 PRINT "That was YES!"
110 STOP
200 PRINT "That was NO!"
210 STOP
```

Fig. 4.12. Adding ELSE to a test of string variables, with a mugtrap incorporated.

tests that are carried out in one line. Line 4Ø starts with IF A$="Y", and normally if A$ is not identical to Y the rest of the line would be ignored. The presence of the word ELSE, however, forces the computer to carry out whatever follows ELSE if, and only if, the first test fails. Let's see how this works.

If A$ is Y, then the first test in line 4Ø succeeds, and the program moves to line 1ØØ. This prints a message, and the program ends because of the STOP instruction. If A$ is N, then the first test in line 4Ø fails, but the presence of ELSE forces the computer to carry out the piece of program that follows ELSE. This is another test, so that if A$ is N, the program jumps to line 2ØØ, prints a different message, and stops.

If both tests fail, though, the program will move from line 4Ø to line 5Ø. Your answer was not exactly Y or N, so that you are asked to try again, and the GOTO 3Ø at the end of line 5Ø causes the program to repeat from line 3Ø. This line constitutes a *mugtrap*, a way of trapping mistakes. Very often when you have a choice of answers, you want to be sure that only certain replies are permitted. A mugtrap is a section of program that is intended to deal with an

incorrect entry. A good mugtrap should show the user the error of his or her ways, and indicate what answer or answers might be more acceptable. This is very often important, because an incorrect entry in some types of program could cause the program to stop with an error message showing. For the skilled programmer (you, by the time you have worked your way through this book), this is just a minor annoyance, but for the inexperienced user it can cause a minor panic. A good program doesn't allow any entries that would cause the program to stop. Mugtraps are our method of ensuring this.

Just to emphasise the sort of power that these simple instructions give you, Fig. 4.13 illustrates a very simple number-guessing game.

```
10 CLS : LET X=INT(RND*10+1)
20 PRINT "Guess the number!"
30 PRINT : PRINT "If you get near, I'll
 tell you"
40 INPUT "Your guess-";N
50 IF N=X THEN  PRINT "Spot on!": STOP
60 IF ABS(N-X)<3 THEN  PRINT "Close - i
t was ";X: STOP
70 GOTO 10
```

Fig. 4.13. A simple number-guessing game which uses number comparisons.

Line 10 clears the screen, and the LET X = INT (RND*10+1) step causes variable X to take a value that lies between 1 and 10. We can't predict what this value will be, because RND means 'select a fraction at random'. A random fraction means a number that is somewhere between 0.000000001 and 0.999999999. Now if we multiply a random fraction by 10, then the number that we get will lie somewhere between 0.00000010 and 9.99999990. Adding 1 to this gives a number that is somewhere between 1.00000010 and 10.99999990. The INT instruction chops off the fraction, and makes the range anything between 1 and 10 – any whole number, that is. RND picks fractions randomly enough for games purposes, but not quite randomly enough for serious statistical users. If you want a more genuinely random number, then any program of this type should start with RAND, or with RAND followed by any negative number, like RAND (−1). Getting back to the program, the instructions in lines 20 and 30 ask you to guess the size of the number, with the difference that you don't have to find it exactly. You enter your number at line 40, and

the tests are made in lines 5Ø and 6Ø. If the number that you picked is identical to the random number, then you get the 'Spot on!' message in line 5Ø, and the program stops. The less obvious test is in line 6Ø. The expression N − X is the difference between your guess, N, and the number X. If your guess is larger than the number, then N − X is a positive number. If your guess is less than X, then N − X is a negative number. The effect of ABS, however, is to make any number positive, so that if X were 5 and you guessed 6 or 4, then ABS (N − X) would come to 1. If you get a difference of 1 or 2 (less then 3), the message in line 6Ø is printed, and the program stops. If you don't get anywhere near, the program repeats, using another random number, because of its GOTO 1Ø in line 7Ø. It's very simple, but quite effective. How about devising a scoring system?

Single key reply

So far, we have been putting in Y or N replies with the use of INPUT, which means pressing the key and then pressing RET. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press RET. For snappier replies, however, there is an alternative in the form of INKEY$. INKEY$ is an instruction that carries out a check on the keyboard to find if a key is pressed. This checking action is very fast, and normally the only way that we can make use of it is by placing the INKEY$ instruction in a loop which will repeat until a key is pressed. Figure 4.14 shows such a loop. The INKEY$

```
10 CLS
20 PRINT "Press any key..."
30 LET K$=INKEY$: IF K$="" THEN  GOTO 3
0
40 PRINT "It was ";K$
50 REM some keys do not give a characte
r
```

Fig. 4.14. Using INKEY$ to find when a key has been pressed. Some keys will cause the program to operate, but will not print anything on the screen.

instruction will produce a string quantity when any key is pressed, so we assign INKEY$ to a string variable, K$. In this way, when any key is pressed, the quantity that it represents will be assigned to K$, and if K$ is a *blank string,* meaning that no key was pressed, the line loops back to its start again. Note how we indicate a blank string by

using two quotes with no space between them. By using the program of Fig. 4.14 you can see the effect of pressing different keys. By changing line 5Ø to GOTO 2Ø, you can make this program repeat until you press the BRK key. In this way, you can find which keys will have the effect. Some keys will produce no visible character on the screen in line 4Ø, but will nevertheless allow the program to jump out of its loop in line 3Ø. Typical of these are the LINE FEED, BS, and RET. Some other keys have no effect, such as the SHIFT, ALPHA LOCK, and CTRL keys. The ESC key has a *very* odd effect – it brings line 4Ø down for editing, and issues an error message 'SE.B', which means that you shouldn't use ESC in this way! Note, by the way, that one key *certainly* won't work in this way – the BReaK key! When you use INKEY$ in a program, then, it's better to print the message 'Press any letter' or 'Press any number'

Menus and subroutines

A choice of two items, such as in Fig. 4.11 isn't exactly a consumer's dream, not in the West anyway. We can extend the choice by a program routine that is called a *menu*. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. We could use a set of lines such as:

```
IF K = 1 THEN GOTO 1ØØØ
IF K = 2 THEN GOTO 2ØØØ
```

and so on. There is a much simpler method, however, which uses a new instructions ON N GOTO, where N is a number variable. You can use any number variable, of course, not just N.

Figure 4.15 shows a typical menu that uses this instruction. Lines 1Ø to 9Ø present the menu items on the screen, and line 1ØØ then invites you to pick one item by typing its numbers. The INKEY$ loop in line 11Ø keeps the program looking for a key until you make your choice, and then line 12Ø tests your choice with a mugtrap. There's a new instruction, VAL, in line 12Ø. VAL means 'number value', and it's used to convert a number that is in string form back into number form. This has to be done because INKEY$ produces a *string* variable, and you can't compare a string with a number (nor a rose with a carrot). By using LET K=VAL (K$) you get a number variable K which will hold a number that is in the correct form to be

```
10 CLS
20 CSR 16,1: PRINT "MENU"
30 PRINT : PRINT
40 PRINT "1. Enter names."
50 PRINT "2. Enter phone numbers."
60 PRINT "3. List all names."
70 PRINT "4. List local numbers."
80 PRINT "5. End program."
90 PRINT
100 PRINT "Please select by number_"
110 LET K$=INKEY$: IF K$="" THEN  GOTO
110
120 LET K=VAL(K$): IF K<1 OR K>5 THEN
PRINT "Incorrect choice - 1 to 5 only":
 PRINT "Please try again": GOTO 110
130 ON K-1 GOTO 150,160,170,180,190
140 STOP
150 PRINT "Names section here": STOP
160 PRINT "Numbers section here": STOP
170 PRINT "List of names here": STOP
180 PRINT "Local numbers here": STOP
190 STOP
```

Fig. 4.15. A menu choice which uses the ON N GOTO instruction.

compared. If you had pressed a letter key then K will be zero.

The choice is then made in line 13Ø, with the ON K−1 GOTO instruction. Now what happens here? If K equals 1, then the first line number that follows GOTO is to be used. The Memotech, unlike any other computer, starts its counting in this instruction with *zero*. That's why we have to use K−1, rather than K. If K=1, then K−1=Ø, and this selects the first of the line numbers that follows GOTO. If K equals 2, then the number that is used is 1, and the second line number following GOTO is used, and so on. All that you have to do is to arrange the line numbers in the same order as your choices. You needn't have a list that looks neat. A line such as ON K−1 GOTO 5Ø,216,484,714,1ØØØ would be just as satisfactory so long as these numbers contained the start of routines that dealt with the menu choices. In this example, the line numbers simply lead to PRINT instructions so as to keep the example reasonably short.

This type of menu selection is useful, but an even more useful method makes use of subroutines. A subroutine is a section of program which can be inserted anywhere that you like in a longer program. A subroutine is inserted by typing the instruction word

GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike GOTO, however, GOSUB offers an automatic return. The word RETURN is used at the end of the subroutine lines, and it will cause the program to return to the point immediately following the GOSUB. Figure 4.16 illustrates this. When the program runs, line

```
10 CLS
20 PRINT "This is a ";
30 GOSUB 1000
40 PRINT "Subroutine": PRINT : PRINT
50 PRINT "Red light and green light mak
e ";: GOSUB 1000: PRINT : PRINT "light.
"
100 STOP
1000 PRINT "yellow ";
1010 RETURN
```

Fig. 4.16. Using a subroutine – this is the key to more advanced programming.

2Ø prints a phrase, with the semicolon used to prevent a new line from being selected. The GOSUB 1ØØØ in line 3Ø then causes the word 'yellow' to be printed, but the RETURN in line 1Ø1Ø will send the program back to line 4Ø, the instruction that immediately follows the GOSUB 1ØØØ. This action will also occur even when the GOSUB is part of a multistatement line, as line 5Ø demonstrates. The GOSUB 1ØØØ will cause the word yellow to be printed, but the RETURN is to the PRINT instructions that follows GOSUB 1ØØØ in line 5Ø, it doesn't jump to line 1ØØ. This example is, of course, a yellow subroutine. The STOP in line 1ØØ is important. Without this, the program would continue on from line 5Ø to carry out the next number, which is 1ØØØ. It would therefore print 'yellow', but then the instruction RETURN has it flummoxed. RETURN where, when there hasn't been a GOSUB? The program stops, with the error message 'No call', meaning that the subroutine was not 'called' by the use of GOSUB 1ØØØ.

Now for something more serious. Figure 4.17 shows subroutines in use as part of an (imaginary) games program. Lines 1Ø to 7Ø offer a choice, and line 9Ø invites you to choose. The familiar INKEY$ and mugtrap actions follow, and then line 12Ø causes the choice to be carried out. This time, however, the program will return to whatever follows the choice. For example, if you pressed key 1, then

```
10 CLS : PRINT
20 CSR 9,1: PRINT "Choose your Monster"
30 PRINT
40 PRINT ,"1. Vampire."
50 PRINT ,"2. Werewolf."
60 PRINT ,"3. Zombie."
70 PRINT ,"4. Shop steward."
80 PRINT
90 PRINT "Select by number, please"
100 LET K$=INKEY$: IF K$="" THEN  GOTO
100
110 LET K=VAL(K$): IF K<1 OR K>4 THEN
PRINT "Faulty selection- 1 to 4 only. P
lease try again.": GOTO 100
120 ON K-1 GOSUB 1000,2000,3000,4000
130 PRINT "That's the END."
140 STOP
1000 PRINT "Red is my favourite colour"
: RETURN
2000 PRINT "I want them all to be like
me": RETURN
3000 PRINT "I follow, master.": RETURN
4000 PRINT "I agree with my brothers.":
 RETURN
```

Fig. 4.17. A menu choice that makes use of subroutines.

the subroutine that starts at line 1ØØØ is carried out, and the program returns to line 12Ø to check if you might also want subroutines 2ØØØ, 3ØØØ, or 4ØØØ. Since the value of K is still 1, the program then goes to line 13Ø and ends. If line 1ØØØ had altered the value of K, however, you could find that a second subroutine was selected following the first one! A subroutine is extremely useful in menu choices, but it's even more useful for pieces of program that will be used several times in a program. Take a look at Fig. 4.18 by way of an example. The subroutine is simply the INKEY$ 'press-any-key' routine, and it's one that you are likely to use many times in the course of any program. Putting the INKEY$ into a subroutine means that you need to type these program lines once only. Wherever you need the action, you simply type GOSUB 1ØØØ (or whatever line number you have used), and the routine will be inserted when the program runs. One problem that you can find is that the program runs too fast for you! In this example, the PAUSE 2ØØ slows the program down by making it wait before carrying out

```
10 CLS
20 PRINT "Choose 1 or 2, please"
30 GOSUB 1000: PAUSE 200
40 LET A=VAL(K$)
50 PRINT "Choose Y or N, please"
60 GOSUB 1000
70 LET B$=K$
80 PRINT "You chose ";A;" and ";B$
90 STOP
1000 LET K$=INKEY$: IF K$="" THEN  GOTO
 1000
1010 RETURN
```

Fig. 4.18. Using INKEY$ for single key reply.

line 4Ø. If you remove the PAUSE, you will usually find that your finger is still on the 1 or 2 key when line 4Ø is executed, so you never get a chance to carry out the action of pressing Y or N! Pause 1ØØØ gives a delay of about a second, though the timing is not precise, and it can depend on what else is being done. Where we simply want a delay, however, it's good enough.

Figure 4.19 shows an elaboration on this one. The trouble with

```
10 CLS
20 PRINT "CHOOSE 1 OR 2, PLEASE"
30 GOSUB 1000
40 LET A=VAL(K$)
50 PAUSE 1000: PRINT "CHOOSE Y OR N, PL
EASE"
60 GOSUB 1000
70 LET B$=K$
80 PRINT "YOU CHOSE ";A;" AND ";B$
90 STOP
1000 LET K$=INKEY$
1010 IF K$<>"" THEN  RETURN
1020 PRINT "*";
1025 PAUSE 250
1030 PRINT CHR$(8);" ";CHR$(8);
1035 PAUSE 250
1040 GOTO 1000
```

Fig. 4.19. A flashing-asterisk subroutine. The asterisk flashes until you press a key.

INKEY$ is that it doesn't remind you that it's in use, there's no question mark printed as there is when you use INPUT. The

subroutine in lines 1ØØØ to 1Ø4Ø remedies that by causing an asterisk to flash while you are thinking about which key to press. The asterisk is flashed by alternatively printing the asterisk and a blank space. There would be no point in printing an asterisk followed by a blank space, so we need to shift the cursor back by one space after each printing action. CHR$(8) is what causes the back-space action, and we'll look at this CHR$ instruction method later. Meantime, make friends with subroutines. They are not just a useful way of obtaining an action at several points in a program, they are an indispensable aid to program planning, of which there's much more in Chapter Eleven.

Chapter Five

Programs with Strings Attached

String functions

In Chapter Three, we took a fairly brief look at number functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them that way is that the really eye-catching and fascinating actions which the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's look at an example. Figure 5.1 shows a program that prints MEMOTECH as a title. What makes it

```
10 CLS
20 FOR N=1 TO 15
30 LET FRAME$(N)="#": NEXT
40 PRINT FRAME$;"MEMOTECH";FRAME$
```

Fig. 5.1. Inserting characters into a string.

more eye-catching is the fact that the word is printed with fifteen hash marks (#) on each side. The hash marks are produced by a string function called *string insertion*, which is something that few computers allow. The way it works is quite simple. When you declare a string variable, the computer allows space for 64 characters, even if you use only a few. You can refer to the position of each character by a number, its place in the string. For this purpose, the first character in the string is number 1 (no, we don't start at zero this time!), the second is 2, and so on. The loop in lines 2Ø and 3Ø of Fig. 5.1 therefore places a hash mark into character positions 1 to 15 of a string called FRAME$. In other words, this string consists of fifteen hash marks! We could make this into a subroutine by using another string name, such as MARK$ in place of the hash mark. The subroutine would then make a fifteen-

character string of whatever character you assigned to MARK$. You could go one step further, and use a number variable in place of the number 15. The subroutine would then make a string of as many characters as you specified. That's how an idea develops into something really useful!

String insertion is a useful way of creating strings of one character, and it's particularly useful when we come to look at graphics characters. There are, however, strings attached, as it were. One is string space. When your Memotech is switched on, it reserves some memory for storing each string. The amount is fairly small, only enough for 64 characters per string, because most programs use far less then this in each string. When you make a lot of use of the string insertion instructions, however, you can sometimes bite deeply into this generous allocation, and this will cause your program to stop with a 'Subscript', or 'No space' message when the allocation is used up. You can reserve more string space by using a DIM instruction. DIM means *dimension,* and if you wanted to provide for a string called FRAME$ which would have 100 characters, you would have a line, near the beginning of your program, which was:

```
10 DIM FRAME$(100)
```

The point about having this near the beginning is that the DIM instruction must be carried out before you start to fill FRAME$ with characters. You are *not* limited to having a maximum of 255 characters in a string, incidentally, as you are with many computers. Providing that you use DIM to prepare for each string that has more than 64 characters, you can have as many as you like! Very few computers allow such long strings to exist.

Code comfort

Each character that is used by the Memotech is represented by a code number, using what we call ASCII code. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced Askey) code is one that is used by most computers. Figure 5.2 shows a printout of the ASCII code numbers and the characters that they produce on my printer (EPSON MX 80). The number characters of ASCII code extend only from 32 to 127. The code numbers above 127 can be used by the Memotech for

32		33	!	34	"
35	#	36	$	37	%
38	&	39	'	40	(
41	)	42	*	43	+
44	,	45	-	46	.
47	/	48	0	49	1
50	2	51	3	52	4
53	5	54	6	55	7
56	8	57	9	58	:
59	;	60	<	61	=
62	>	63	?	64	@
65	A	66	B	67	C
68	D	69	E	70	F
71	G	72	H	73	I
74	J	75	K	76	L
77	M	78	N	79	O
80	P	81	Q	82	R
83	S	84	T	85	U
86	V	87	W	88	X
89	Y	90	Z	91	[
92	\	93	]	94	^
95		96	`	97	a
98	b	99	c	100	d
101	e	102	f	103	g
104	h	105	i	106	j
107	k	108	l	109	m
110	n	111	o	112	p
113	q	114	r	115	s
116	t	117	u	118	v
119	w	120	x	121	y
122	z	123	{	124	!
125	}	126	~	127	

Fig. 5.2. The standard ASCII code numbers.

other purposes, and we can select how we make use of them. This is something that we'll investigate in more detail in Chapter Six.

The logic of LEN

String variables allows us to carry out a lot of operations that can't be done with number variables. One of these operations is finding out how many characters are contained in a string. Since a string can contain up to 64 characters (without using DIM), or more if DIM is

used, a method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable, within brackets, and the result of using LEN is always a number so that we can print it or assign it to a number variable.

Figure 5.3 shows a simple example of LEN in use. Line 2Ø assigns

```
10 CLS
20 LET TITLE$="MEMOTECH"
30 PRINT "There are ";LEN (TITLE$);" ch
aracters in ";TITLE$
```

Fig. 5.3. Introducing LEN, a member of the string function family.

a variable and line 3Ø tells you how many letters are in this variable. Note that there *must* be a space between LEN and the bracket that encloses the string. This action is hardly earth-shattering, but we can turn it to very good use, as Fig. 5.4 illustrates. This program uses LEN as part of a subroutine which will print a string called TITLE$ centred on a line. This is an extremely useful subroutine to use in

```
10 CLS : LET Y=1
20 LET TITLE$="MEMOTECHNOLOGY"
30 GOSUB 1000
40 LET Y=2
50 FOR N=1 TO LEN (TITLE$): LET TITLE$(
N)="*": NEXT
60 GOSUB 1000
70 STOP
1000 LET X=(39-LEN (TITLE$))/2-1
1010 CSR X,Y: PRINT TITLE$
1020 RETURN
```

Fig. 5.4. Using LEN to print titles centred.

your own programs, because its use can save you a lot of tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string TITLE$. This number is then subtracted from 39, the result divided by two, and 1 subtracted, using the formula that we saw first in Chapter Two. If the number of characters in the string is an odd number, then X will contain a .5, but this is completely ignored by CSR instruction when the string is printed.

The whole process can be done in three lines, in this case lines 1ØØØ to 1Ø2Ø of the subroutine. Once in place, we can call this subroutine to centre anything that has the name TITLE$. In line 2Ø,

TITLE$ is assigned to the word MEMOTECHNOLOGY, and this phrase is printed centred. In line 5Ø, TITLE$ is assigned to a string of as many asterisks as there are letters in the word! This also is done by using LEN to count the letters in TITLE$. This set of asterisks also is printed centred by the subroutine to form an underlining.

Notice, by the way, that if we want anything printed centred by this subroutine, we have to give it the variable name of TITLE$. This action is called *passing a variable* to the subroutine, and it's something that we have to keep a careful eye on when we use subroutines. You can't expect a subroutine that is written to print TITLE$ centred to have any effect on a string called A$.

By the left, slice!

The next group of string operations that we're going to look at are called slicing operations. The result of slicing a string is another string, a piece copied from the longer string. Such an operation does not, however, destroy the original string, since it is a copying action only. String slicing is a way of finding what letters or other characters are present at different places in a string. The Memotech is unique in allowing you two ways of slicing strings. One of these ways is identical to the method that is used by most other computers, the other is a method that is used (though not in an identical way) also by Atari and the ZX machines. Only Memotech has both!

All of that might not sound terribly interesting, so take a look at Fig. 5.5. The strings WORD1$ and WORD2$ are assigned in line

```
10 CLS
20 LET WORD1$="Memory": LET WORD2$="tec
hnology"
30 LET TITLE$=LEFT$(WORD1$,4)+LEFT$(WOR
D2$,4)
40 PRINT TITLE$
50 CSR 2,6
60 PRINT WORD1$(1,4)+WORD2$(1,4)
```

Fig. 5.5. Using the string slicing action LEFT$ and its simpler equivalent.

2Ø, and sliced in line 3Ø, with the result of the slicing being assigned to TITLE$. What's printed in line 4Ø is the word Memotech. Now how did this happen? The instruction LEFT$ means 'copy part of a string starting at the left-hand side'. LEFT$ has to be followed by two quantities, within brackets and separated by a comma. The first

of these is the variable name for the quantity that we want to slice, WORD1$ or WORD2$ in this example. The second is the number of characters that you want to slice (copy, in fact) from the left-hand side. The effect of LEFT$ (WORD1$,4) is therefore to copy the first four letters from Memory, giving Memo. The last part of line 3Ø adds the 'tech' from 'technology' to the first four sliced letters, so giving us the name of our favourite computer printed on the screen in line 4Ø.

Now LEFT$ is an instruction that will be found on all computers that use a variety of BASIC that is called Microsoft, but the Memotech also allows another, simpler, method. This is illustrated in line 6Ø. Instead of using instructions like LEFT$, we simply specify the place number in the string where we want to start slicing, and the number of characters that we want to slice. For both WORD1$ and WORD2$, we want to start at character 1, and slice the first four characters, so we need to use (1,4) following the string name. The advantage of this second method is that it is much more flexible, as you'll see in a moment.

For a more serious use of the left-slicing instruction, take a look at Fig. 5.6. This has the effect of extracting your initials from your

```
10 CLS : CSR 2,2
20 INPUT "Your surname, please? ";SUR$
30 INPUT "Your first name, please? ";FN
M$
40 CSR 1,6
50 PRINT "You'll be known as ";LEFT$(FN
M$,1);".";LEFT$(SUR$,1);".";" around he
re."
```

Fig. 5.6. Extracting initials with LEFT$ string slicing.

name, and it's done by using LEFT$ along with a bit of concatenation. The INPUT steps in lines 2Ø and 3Ø find your surname and forename, and assign them to variable names SUR$ and FNM$. We can't use the more obvious FOR$ for forename, because FOR is a reserved word in BASIC, part of the FOR...NEXT set. Line 5Ø then prints your initials by using LEFT$ to extract the first letter of each string. The letters are then assembled along with full stops, using concatenation in line 5Ø. If you have two players in a game, it's often useful to show the initials and score rather than printing the full name, but the full names can be held stored for use at various stages in the game.

All right, Jack?

String slicing isn't confined to copying a selected piece of the left-hand side of a string. We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the LEFT$ one, but it's useful none the less. Figure 5.7 illustrates the use of the instructions to avoid having to type a word over again. This shows the use of RIGHT$ in line 40. As before, the

```
10 CLS
20 LET WORD$="Memotechnology"
30 CSR 2,6
40 PRINT "It's all high ";RIGHT$(WORD$,
10)
50 CSR 2,9
60 PRINT "- computing ";WORD$(5,10)
```

Fig. 5.7. Using RIGHT$ and its simpler equivalent to extract letters from the right-hand side of a string.

instruction word RIGHT$ has to be followed by a bracket, then the string that is to be sliced, and then the number of places counted from the *right*-hand side of the string. Line 60 shows the alternative method. The place number of the first letter (t) is 5, and there are ten letters, so that WORD$(5,10) will produce the word that we want. There's no need for any LEFT$ or RIGHT$ when you use this type of slicing. There are more serious uses for right-slicing than this example. You can, for example, extract the last four figures from a string of numbers like 010-242-7016. I said a *string* of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type LET N = 010-242-7016 then the computer assumes that you want to subtract 242 from 010 and 7016 from that result. The value for N is −7248, which is not exactly what you had in mind! If you use LET N$ = "010-242-7016" then all is well.

Now we can get quite a lot of interesting effects from LEFT$ and RIGHT$, and their equivalents. Take a look at Fig. 5.8, for an example, which does odd things with the letters of your name. The program prompts you to enter your name in line 20, and the name is assigned to NAME$. In line 30, we use LEN so that the number variable L contains the total number of characters in your name. This will include spaces and hyphens – nobody's likely to use asterisks and hash marks! Line 40 then starts a loop which uses the

```
10 CLS
20 CSR 2,2: INPUT "Your name, please ";
NAME$
30 LET L=LEN (NAME$)
40 FOR N=1 TO L
50 CSR 1,N+2: PRINT LEFT$(NAME$,N);: CS
R 20,N+2: PRINT RIGHT$(NAME$,N)
60 NEXT
```

Fig. 5.8. Slicing both left and right sides of a string.

total number of characters as its end limit. Line 5Ø is the action line. When N is 1, line 5Ø prints the first letter on the left of your name on to the left-hand side of the screen, and the first letter on the right of your name on the right-hand side. The positioning is done by using CSR. On the next pass through the loop, a new line is selected, and two letters are printed. This continues until the entire name is printed. If you use a LEFT$ or RIGHT$ with a number that is more than the number of letters in the string, incidentally, then you simply get the whole string. How about converting this program so that it uses the other type of slicing command?

Pig in the MIDdle?

There's another string slicing instruction which is capable of much more than either LEFT$ or RIGHT$. The instruction word is MID$, and it has to be followed by three items, within brackets, and using commas to separate the items. Item 1 is the name of the string that you want to slice, as you might expect by now. The second item is a number which specifies where you start slicing. This number is the number of the character counted from the left-hand side, and counting the first character as 1. The third item is another number, the number of characters that you want to slice, going from left to right and starting at the position that was specified by the first number.

It's a lot easier to see in action than to describe, so try the program in Fig. 5.9. Line 2Ø assigns TITLE$ to "Memotech genius", and line 3Ø finds L, the number of characters in the title. The loop that starts in line 4Ø then prints letters taken from the words "Memotech genius". With the value of N equal to 1, the letter that is sliced is M, because its position in the word is 1, and we're copying one letter from this position. If we used MID$(NAME$,1,2), we would get

```
10 CLS
20 LET TITLE$="Memotech genius"
30 LET L=LEN (TITLE$)
40 FOR N=1 TO L
50 PRINT MID$(TITLE$,N,1);" ";: NEXT
60 CSR 0,5
70 FOR N=1 TO L
80 PRINT TITLE$(N,1)+"+";: NEXT
```

Fig. 5.9. Using MID$ and its equivalent. This can extract from any part of a string, and can, like LEFT$ and RIGHT$, be controlled by variables.

'Me', and if we used MID$(NAME$,3,2) we would get 'mo'. As it is, we select a letter at a time, and print a space. The semicolon in line 5Ø then ensures that the next sliced letter is printed on the same line. The net effect is that the letters are printed spaced out. The second loop in lines 7Ø and 8Ø performs the same kind of effect. This time, though, it uses the alternative instruction, which doesn't need the MID$ part, and it places a + sign between the letters rather than a space.

One of the features of all of these string slicing instructions is that we can use variable names or expressions in place of numbers. Figure 5.10 shows a more elaborate piece of slicing which uses expressions. It all starts innocently enough in line 2Ø with a request

```
10 CLS
20 CSR 2,2: INPUT "Your name, please ";
NAME$
30 LET L=LEN (NAME$): LET C=INT(L/2)+1
40 FOR N=1 TO C
50 CSR 20-N,N+3: PRINT NAME$(C-N+1,N*2-
1)
60 NEXT
```

Fig. 5.10. Making a letter pyramid to show the action of MID$ with a formula.

for your name. Whatever you type is assigned to variable NAME$, and in line 3Ø a bit of mathematical juggling is carried out. How does it work? Suppose you type as your name 'DONALD'. This has six letters, so in line 3Ø L is assigned to 6, and C is the whole number part of L/2 (equal to 3), plus 1, making 4. Line 4Ø then starts a loop of four passes. In the first pass, with N = 1, you print at the position that is given by CSR 19,4 (because N is 1, and 2Ø−N is 19, with N+3=4). What you print is the part of the name that is sliced by NAME$(4 − 1 + 1,2 − 1), which is NAME$(4,1), the single letter in

place 4 which is A in this example. On the next run through the loop, N is 2, C − N + 1 is 3, and N*2 − 1 is also 3. What is printed is MID$(NM$,3,3), which is NAL. The loop goes on in this way, and the result is that you see on the screen a pyramid of letters formed from your name. It's quite impressive if you have a long name. If you don't have a long name, try making one up!

Some odd (and even) characters

It's time now to look at some other types of string functions. We've met VAL previously – it's used to convert a number that is in string form back into number form so that we can carry out arithmetic. VAL will work only as far as the first non-number character it meets. For example, if A$="23456", then VAL (A$) gives 23456, but if A$ = "23 Fifth and 76" then VAL (A$) gives 23. Any numbers that *follow* letters are ignored, so that if A$ = "We2", then VAL (A$) gives ∅.

There's an instruction that does the opposite conversion, STR$. When we follow STR$ by a number, number variable, or expression within brackets, we carry out a conversion to a string variable. We can then print this as a string, or assign it to a string variable name, or use string functions like LEN, MID$ and all the others. Figure 5.11 illustrates these processes – with a warning! Lines 1∅ to 3∅ show that

```
10 LET N$="22.5": LET V=2
20 CLS : PRINT
30 PRINT N$;" times ";V;" is ";V*VAL(N$
)
40 PRINT
50 LET V$=STR$(V)
60 PRINT " There are ";LEN (V$);" chara
cters in ";V;" !"
70 PRINT
80 PRINT N$;" added to ";V$;" gives ";N
$+V$;" ?"
```

Fig. 5.11. Using STR$ and VAL for converting between string and number form. Note the space that STR$ puts in.

we can do arithmetic on N$ if we use VAL with it. Line 5∅ converts the number variable V into string form. Now, V has been assigned to the number 2 in line 1∅, and we would expect just one character to be present in the string. Line 6∅ reveals that there are two! The reason is

that when we use STR$ to convert a number into string form, a space is left at the left-hand side of the string in case we want to put in a sign, + or −. This space is, of course, an extra character, which explains why 2 appears to consist of two characters, and 42 would consist of three characters. Line 8Ø shows the strings being concatenated, just to emphasise the difference between string variables and number variables.

If you hark back to Fig. 5.2, now, you'll remember that we introduced the idea of ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by the function ASC, which is followed, within brackets, by a string character. The result of ASC is a number, the ASCII code number for that character. If you use ASC ("MEMOTECH"), then you'll get the code for the M only, the action of ASC includes rejecting more than one character. Figure 5.12 shows this in action. String variable A$ is assigned in

```
10 LET A$="MEMOTECH"
20 CLS : PRINT
30 FOR N=1 TO LEN (A$)
40 PRINT ASC(A$(N,1));" ";
50 NEXT
```

Fig. 5.12. Using ASC to find the ASCII code for letters.

line 1Ø and in line 3Ø a loop starts which will run through all the letters in A$. The letters are picked out one by one, using string slicing, and the ASCII code for each letter is found with ASC. The space between quotes, along with the semicolons in line 4Ø make sure that the codes are all printed on one line with a space between the numbers. Simple, really.

ASC has an opposite function, CHR$. What follows CHR$, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction PRINT CHR$(65), for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for coding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not too obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.13 illustrates this use. Line 5Ø contains an INKEY$ loop to make the program wait for you. When you press a key, the loop that starts in line 7Ø prints 29

```
10 CLS : PRINT
20 PRINT "What does MTX mean?"
30 PRINT
40 PRINT "Press any letter key to revea
l all."
50 LET K$=INKEY$: IF K$="" THEN  GOTO 5
0
60 PRINT
70 FOR J=1 TO 29: READ N
80 PRINT CHR$(N);
90 NEXT
100 STOP
110 DATA  77,97,120,105,109,117,109,32,
84,101,99,104,110,105,99,97,108,32,101,
88,99,101,108,108,101,110,99,101,33
```

Fig. 5.13. Using ASCII codes to carry a coded message, and then using CHR$ to obtain the character that corresponds to a code number.

characters on the screen. Each of these is read as an ASCII code from a list, using a READ...DATA instruction in the loop. The PRINT CHR$(N) in line 8Ø then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 20, or anything else you liked. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent decoders!

The law about order

We saw earlier in Fig. 4.10, how numbers can be compared. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign (=) means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 5.14 illustrates this use of comparison using the = and > symbols. Line 2ϕ assigns a nonsense word – it's just the first six letters on the top row of letter

```
10 CLS
20 LET A$="QWERTY"
30 PRINT : INPUT "Type a word ";B$
40 IF B$=A$ THEN  PRINT "Same as mine!"
: STOP
50 IF A$>B$ THEN  LET C$=A$: LET A$=B$:
 LET B$=C$
60 PRINT "Correct order is ";A$;" then
";B$
```

Fig. 5.14. Comparing words to decide on their alphabetical order.

keys. Line 3ϕ then asks you to type a word. The comparisons are then carried out in lines 4ϕ and 5ϕ. If the word that you have typed, which is assigned to B$ is *identical* to QWERTY, then the message in line 4ϕ is printed, and the program ends. Remember that qwerty is *not* identical to QWERTY! If QWERTY would come later in an index than your word, then line 5ϕ is carried out. If, for example, you typed PERIPHERAL, then since P comes earlier than Q in the alphabet and has an ASCII code that is less than the code for Q, the word A$ scores higher than B$, and line 5ϕ swaps them round. This is done by assigning a new string, C$ to A$ (so that C$ = "QWERTY"), then assigning A$ to B$ (so A$ = "PERIPHERAL"), then B$ to C$ (so that B$="QWERTY"). Line 6ϕ will then print the words in the order A$ and then B$, which will be correct alphabetical order. If the word that you typed comes later than QWERTY, for example, TAPE, then A$ is not greater than B$, and the test in line 5ϕ fails. No swap is made, and the order A$, then B$, is still correct. Note the important point though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts. One point to watch is when you combine the = sign with the < or > signs. The = sign *must* come second in such a combination, otherwise the computer takes it that something is being assigned to a variable name of < or >!

Put it on the list

The variable names that we have used so far are useful, but there's a

limit to their usefulness. Figure 5.15 illustrates this. Lines 2Ø to 4Ø generate an (imaginary) set of examination marks. This is done simply to avoid the hard work of entering the real thing! The variable in line 3Ø is something new, though. It's called a *subscripted*

```
10 CLS : DIM A(10)
20 FOR N=1 TO 10
30 LET A(N)=INT(1+100*RND)
40 NEXT
50 PRINT
60 CSR 13,2: PRINT "MARKS LIST"
70 PRINT : FOR N=1 TO 10
80 CSR 2,N+3: PRINT "Item ";N;" receive
d ";A(N);" marks"
90 NEXT
```

Fig. 5.15. An array of subscripted number variables. It's simpler than the name suggests!

variable, and the subscript is the number that is represented by N. The name that we use has nothing to do with computing, it's a name that was used long before computers were around. How often do you make a list with the items numbered 1, 2, 3... and so on? These numbers 1, 2, 3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. A member of this group like A(2) has its name pronounced as 'A-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity numbers. Since the number can be a number-variable or an expression, this allows us to work with any item of the list. Figure 5.15 shows the list being constructed from the FOR...NEXT loop in lines 2Ø to 4Ø. Each item is obtained by finding a random number between 1 and 1ØØ, and is then assigned to A(N). Ten of these 'marks' are assigned in this way, and then lines 6Ø to 9Ø print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

This convenience has to be paid for by some preparation, however, We have to decide in advance how many subscript numbers we are going to use, and prepare enough memory space to hold them. As you might expect, this is the action of DIM A(1Ø) in line 1Ø. Using DIM A(1Ø) allows you to use variables from A(Ø) to A(1Ø), which is, in

fact, eleven subscripted variables. If you forget to 'dimension the variable', then you will get the 'Undefined' error message at the first line which makes use of the variable name, line 3Ø in this example.

Figure 5.16 extends this another step further. This time you are invited to type a name and a mark for each of ten items. After you

```
10 CLS : PRINT : DIM A(10),NAME$(10,20)
20 PRINT "Please enter names and marks"
30 FOR N=1 TO 10
40 CSR 3,6: PRINT "Name ";: INPUT NAME$
(N)
50 CSR 3,8: PRINT "Mark ";: INPUT A(N)
60 CSR 3,6: PRINT CHR$(5): CSR 3,8: PRI
NT CHR$(5)
70 REM chr$(5) is 'erase to end of line
'
80 NEXT
90 CLS : LET TOTAL=0
100 CSR 13,0: PRINT "MARKS LIST": PRINT

110 FOR N=1 TO 10
120 CSR 2,N+3: PRINT NAME$(N): CSR 20,N
+3: PRINT A(N)
130 LET TOTAL=TOTAL+A(N)
140 NEXT
150 PRINT
160 PRINT "Average is ";TOTAL/10
```

Fig. 5.16. Using strings in one array, and numbers in another.

have pressed RET following the mark entry, the part of the screen where you entered the name and mark is cleared by line 6Ø. This is done because the effect of PRINT CHR$(5) is to erase the end of the line, and the CSR instruction has placed the cursor at the start of each of the lines in turn. When the list is complete, the screen is cleared and a TOTAL variable is set to zero in line 9Ø. The list is then printed neatly, and on each pass through the loop the total is counted up (in line 13Ø) so that the average value can be printed at the end. The important point here is that it's not just numbers that we can keep in this list form. The correct name for the list is an *array*, and Fig. 5.16 uses both a string array (names) and a number array (marks).

Once again, you have to prepare for the use of an array by the use of DIM. The form of DIM that you need to use for a string array,

however, is quite different from the form that you use for numbers. As usual, brackets follow the name of the variable, but there are two numbers within the brackets. The first of these is the maximum subscript number, just as you would use for a number array. The second is the maximum number of characters in each string. In the example, this number is 2Ø. We would not, therefore, be able to use any name of more than 20 letters unless this number was changed. It may seem a bit of a limitation, but it allows the computer to work very much faster with arrays of string characters. You can, of course, use names with fewer than 20 letters, and arrays with fewer than 10 subscript numbers even though you have used DIM NAME$(1Ø,2Ø).

Rows and columns

You can imagine an array as a list of items, one after the other, but there is a variety of array which allows a different kind of list, called a *matrix*. A matrix is a list of groups or items, with all the items in a group related. We could think of a matrix as a set of rows and columns, with each group taking up a row, and the items of a group in separate columns. Take a look at Fig. 5.17 to see how this works.

```
10 CLS : DIM NAME$(3,2,10)
20 FOR ROW=1 TO 3
30 FOR COLUMN=1 TO 2
40 READ NAME$(ROW,COLUMN)
50 NEXT COLUMN: NEXT ROW
60 FOR ROW=1 TO 3
70 CSR 5,ROW+3: PRINT NAME$(ROW,1): CSR
 25,ROW+3: PRINT NAME$(ROW,2)
80 NEXT
100 DATA  horse,foal,cow,calf,dog,pup
```

Fig. 5.17. Making a matrix of rows and columns. Note the DIM method.

We use here a variable NAME$ which has two subscript numbers. The first number is the row number, the second is the column number, and we need two FOR...NEXT loops to read data into this matrix. This is carried out in lines 2Ø to 5Ø. The items are then printed in columns by the loop in lines 6Ø to 8Ø. In this loop, the variable ROW is used as the row number and we use the column numbers 1 and 2. The rows contain animal names, and the columns separate the different names that we use for adult and for young animals respectively. Notice how we have to dimension this, by using

three numbers. These are the row and column numbers, and then the maximum number of letters in each name. Note, incidentally, how the first item of data has a space placed in front of it. This is a minor problem with the DATA part of READ DATA, and there doesn't seem to be any simple remedy. If your data item starts with a semicolon, however, the space is omitted!

Figure 5.18 shows a much more ambitious matrix program. This

```
10 CLS : DIM NAME$(50,2,30)
20 FOR N=1 TO 50
30 CSR 2,3: PRINT "Name ";: INPUT NAME$
(N,1)
40 CSR 2,5: PRINT "Tel. No.  ";: INPUT
NAME$(N,2)
50 CSR 2,3: PRINT CHR$(5)
60 CSR 2,5: PRINT CHR$(5)
70 NEXT
80 CLS : PRINT "List complete."
90 PRINT "Pick an initial."
100 INPUT J$
110 FOR N=1 TO 50
120 IF J$=LEFT$(NAME$(N,1),1) THEN  GOS
UB 500
130 NEXT
140 GOTO 90
500 PRINT "Name - ";NAME$(N,1)
510 PRINT : PRINT "Number - ";NAME$(N,2
)
520 RETURN
```

Fig. 5.18. Using a name and number matrix for a simple telephone directory application. This can be saved after running, so that the values can be used next time the program is loaded.

one is dimensioned as usual in line 1Ø. The idea is to store sets of names and telephone numbers which are fed in by you in the course of the loop in lines 2Ø to 7Ø. Once the matrix has been filled, you can pick an initial letter for a name, and ask the computer to print out the name and number that it has located. I've left out mugtraps just to keep this example reasonably short, but you would certainly need some sort of mugtraps, even if only in the form of a message like:

```
135 PRINT" SORRY, CAN'T FIND ";J$;" ENTRIES"
```

A small rearrangement of this program produces another benefit. As it stands, you need to press BRK to get out of the loop. If you

arrange it so that entering X as an initial will do this, you can make the program go to another line. This line must follow a line with the instruction STOP, so that it cannot be carried out in any part of the program, except by a GOTO, or by entering a 'terminator', like 'X'. If this line, perhaps line 1ØØØ, looks like this:

```
1ØØØ SAVE "Names"
1Ø1Ø GOTO 9Ø
```

then when you stop using the program, all of its variables will be preserved. This program will take a noticeably longer time to SAVE than its equivalent with no SAVE instruction in the program. You must, of course, give yourself time to prepare the cassette recorder, so a subroutine which prints a message and waits until you press a key would be useful. When you LOAD this version next time, it will start at line 9Ø, ready to replay the values for you! This is the way in which the Memotech stores variable values. It would be even more useful if the variable values could be stored separately from the program, but this is possible only with the disk machines.

Chapter Six
Special Effects

Any modern computer is expected to be able to produce dazzling displays of colour and other special effects. The Memotech is no except, and in this chapter, we'll start to look at some of the effects that are possible. To start with, we have to know some of the terms that are used, and the first of these is *graphics*. Graphics means pictures that can be drawn on the screen, and all modern computers have instructions that allow you to draw such patterns. In connection with these patterns, you'll see the words *low resolution* and *high resolution* used. Resolution isn't such an easy term to explain. Imagine that you are creating pictures on a paper sheet of about eleven inches across by eight inches deep – that's roughly the size of a TV screen that is described as being a 14 inch screen (it's about 14 inches diagonally!).

Now, if you are asked to create the pictures by using rectangles of coloured paper, you are dealing with picture-making in a way that is very similar to the way that the computer operates. Suppose that you are allowed only 936 pieces of paper, of such a size that all 936 will fill the screen. You couldn't draw very finely detailed pictures with so few large pieces, and this is what we mean by low resolution. On the other hand, if you were provided with pieces so small that you would need 49152 of them to fill an entire screen size, you could produce very much more detailed pictures. This is what we mean by high resolution. The Memotech has high resolution graphics commands available, and the figure of 49152 pieces that I have used corresponds to the size of the blocks that the Memotech can use. To create these graphics pictures, a computer has to make use of memory. A lot of computers make use of the same memory as the owner uses for programs. Because of this, a lot of computers that are said to have 64K or 32K of memory can turn out to have very little left for you to use, perhaps as little as 6K! This is because so much of the memory is being used in servicing the graphics display, and the

other operating systems. The Memotech, like the old TRS-80, does not steal your program memory for graphics. Instead, it has a reserved (dedicated) piece of memory that is used for graphics, and the 32K or 64K (or more) of program memory is not grabbed for graphics.

Vivid impressions

The best place to start on our exploration of special effects is with the text modifiers. As the name suggests, these cause changes in the appearance of any text that is printed on the screen. Take a look at the program in Fig. 6.1. There are two new commands here, PAPER

```
10 CLS : CSR 5,2
20 PRINT "this is a colour example."
30 PAPER 8: INK 11
40 PAUSE 5000
50 PAPER 15: INK 3
60 GOTO 60
```

Fig. 6.1. Using the PAPER and INK instructions.

and INK. The names tell you just about all you need to know. PAPER is used to set the colour of the background, and INK is used to set the colour of the letters. What the names don't tell you, however, is what numbers must follow them, and how they affect the appearance of the screen. As line 3Ø of Fig. 6.1 shows, PAPER or INK must be followed by number, in the range of Ø to 15. Figure 6.2 is a list of these numbers. The values of PAPER and INK that you choose for text in this way will affect the whole screen. Line 5Ø of Fig. 6.1 proves this by changing the values after a PAUSE.

Now these commands are acting on what is called the *text screen,* which is the normal arrangement of screen that we use. This text screen can be used only for text, meaning characters that are put into place by the PRINT instruction. As we shall see, there are other instructions that can be used with *graphics screens,* and we'll be looking at these later. Meantime, look at line 6Ø. This is an *endless loop*, and it's there for a very good reason. Whenever a program ends, the screen layout goes back to its familiar pattern of text only, nineteen lines of listing, four line for entry, one line for messages. It also goes back to its setting of blue paper and white ink. If you want to keep text displayed in other colours, then you have to make sure that the program doesn't end! Line 6Ø ensures this.

Number	*Colour*
∅	Transparent (background colour)
1	Black
2	Medium Green
3	Light Green
4	Dark Blue
5	Light Blue
6	Dark Red
7	Cyan (blue plus green light)
8	Medium Red
9	Light Red
10	Dark Yellow (Gold)
11	Light Yellow
12	Dark Green
13	Magenta (red plus blue light)
14	Grey
15	White

Fig. 6.2. The COLOUR numbers to use with PAPER, INK and other instructions.

Create your own characters!

The Memotech offers an interesting way of producing graphics, however, on the ordinary text screen, using only the PRINT instruction to place the patterns. These could be classified as low resolution in the sense that they use the same limited number of PRINT positions on the screen, but they offer much more scope for dazzling effects. As this title suggests, we can create our own character shapes. There are two parts to this – the planning of shapes, and how we place the shapes on the screen. Let's take these two in easy stages.

We'll start, logically enough, with planning. The size of the shape we're talking about is one screen character, the size of the cursor block. Now this, and every other Memotech character, is made out of up to 35 dots arranged on a six by eight grid. Figure 6.3 shows the shape of this grid – you can re-draw it for yourself on a sheet of graph paper if you want more copies. The important point is that the small squares of the grid represent dots on the screen that can be in either INK or PAPER colour, according to the value of code numbers that we use to instruct the computer. When a character is

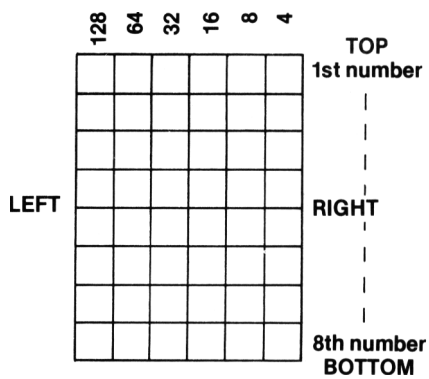


Fig. 6.3. A six by eight planning grid for user-defined characters.

designed on this six by eight grid, we normally use only 5 dots across and 7 down, leaving the right-hand column and the bottom row unused. This is because we want to have a space between any two characters, and also between any two lines of text on the screen. If you are designing your own graphics shapes, however, you might want to make them fill the whole six by eight block, and so join on to each other when you print them on to the screen.

Now, the Memotech manual shows you very briefly how to design these shapes, but unless you have done it before, you may be rather puzzled. The key to it is the numbers that are printed on top of each column of squares. Each number is a code for any square in the column underneath it. Use the number, and the square will be in INK colour. Use ∅ instead, and the square is in PAPER colour, which means invisible. An example will help to make this clearer, and it appears in Fig. 6.4. I've used a simple shape of a 'space-walker' to illustrate the principle, and Fig. 6.5 shows the planning step that is needed.

The first line of squares has just two squares shaded in. I usually

```
10 CLS : PAPER 2: INK 11
20 GENPAT 0,35,48,120,48,252,180,48,72,
132
30 FOR N=1 TO 10
40 LET X=INT(1+RND*39): LET Y=INT(1+RND
*18)
50 CSR X,Y: PRINT "#"
60 NEXT
70 GOTO 70
```

Fig. 6.4. Using the GENPAT instruction to define a character shape.

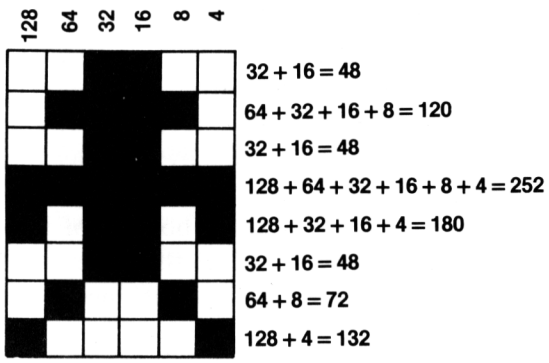


Fig. 6.5. How a space-walker shape is planned.

work on tracing paper clipped over the grid pattern, but in this example, I've shown what it will look like on the graph paper itself. The shaded squares in the top line are the ones that we want to appear in INK colour, and they are under the code numbers 32 and 16. There's nothing else shaded in this line, so we add the numbers 32 and 16, to get 48, which is the number we note at the side. Similarly in the second line down, the squares in the 64, 32, 16 and 8 positions are shaded, and so the number that we use is the sum of these, 120. We continue in this way until all the eight lines have been dealt with.

There are a few points to note. One is that if none of the squares in a line is shaded, the code number is zero. The other point is that you can save a lot of arithmetic by remembering that a complete set of shaded squares in a line adds up to 252. This is the maximum size of a number that you can use as a code number for a text character. You may wonder why the numbers stop at 4. The reason is that *graphics* characters can use 8 columns, and the missing numbers 2 and 1 are used when we plan such characters. We'll deal with that later! You must always end up with eight numbers, no matter what shape you are trying to produce.

The next matter is how we instruct the computer to produce the shape. What we have to do is to store the code numbers in the Memotech's memory, along with an ASCII code number that we will use to obtain the shape on the screen. This makes use of a new instruction, GENPAT. GENPAT (short for GENerate PATtern) has to be followed by ten numbers. The first number is a *mode* number, which dictates how GENPAT will be used. At the moment, we want to create shapes for ASCII codes 32 to 127, and the mode number for this is Ø. The other numbers are used when we want to

make use of GENPAT for other purposes. The second number is the ASCII code that we want to use. In this way, pressing a key or using this ASCII code will give our own pattern. The next eight numbers in GENPAT are just the pattern numbers that we have already found. All of the numbers are separated by commas, there must be a space between the 'T' of GENPAT and the first number, and the pattern numbers are read from top to bottom of the shape. The effect of GENPAT is therefore to place the code number into the memory. We have used 0 as the mode number, because we are going to use one of the ASCII codes in the range 32 to 127. The second number is 35, the code for the hash sign (pound sign on the keyboard), and the other numbers are the set that we noted in Fig. 6.5.

Getting back to Fig. 6.4, then, line 10 sets the PAPER and INK colours, and line 20 contains the GENPAT instruction that creates the shape. Lines 30 to 60 then make ten characters appear on the screen. Line 50 is the clue here, but all is not what it seems. The instruction, 'PRINT "#" ' prints the character that belongs to the hash mark key (the pound sign on the Memotech keyboard), but this character does *not* appear on the screen as a hash mark now. When the program runs, this character prints as the shape that we designed. Line 40 generates some random numbers to use in the CSR instruction so that the character can be printed at random positions on the screen. Since the program uses an endless loop in line 70 to ensure that the colours stay the way they were set in line 10, you have to use BRK to get out of it. Even after this, though, pressing SHIFT 3, the pound sign key, will give the space-walker character, *not* the hash sign.

Now we have the character, what do we do with it? Figure 6.6 shows one possibility. Lines 10 and 20 are the same as before, but line

```
10 CLS : PAPER 2: INK 11
20 GENPAT 0,35,48,120,48,252,180,48,72,
132
25 CSR 2,10: PRINT "MATTHEW SPACEWALKER
 FALLS EARTHWARDS."
30 FOR Y=0 TO 23
40 CSR 18,Y: PRINT CHR$(35);
50 PAUSE 100: CSR 18,Y: PRINT " ";
60 PAUSE 100: NEXT
```

Fig. 6.6. Animating the space-walker. Better animation methods are discussed in Chapter Eight.

25 generates a line of text. Lines 3ø to 6ø then generate a 'falling' character. The loop uses values of Y which will be used by CSR to print the character at different vertical positions. In each position, the character is printed, and then there is a pause. Then there is a 'wiping' action which is done by printing a blank, and another pause follows. This wiping action will also remove one letter from the line of text as the shape falls through the text!

It's not a brilliantly satisfactory animation, but this may be all that you might want in a text program, just to liven things up a bit. Very much better animation is possible by using *sprite* shapes, and we'll look at that topic later, in Chapter Eight. In the meantime, note that the program in Fig. 6.6 has used the alternative method of printing the shape, using CHR$(35). If you want to use these *user-defined* characters in addition to the full complement of ordinary text characters, there are several spare ASCII codes that are available. These are 91 to 96, and 123 to 127, both inclusive.

You are not confined to creating just one character in this way, and you can also create characters that fit together to form a shape! Suppose, for example, that we made new characters to fit the codes for #, $, %, & and '. We could then use the instruction: LET MULTI$="#$%&'", and then the instruction PRINT MULTI$ would print the whole shape! You might think that GR$ would be a good name for a graphics string, but you can't use it – it's reserved for other purposes, as we shall see. Figure 6.7 shows a shape drawn on to five grids stacked together. Figure 6.8 then shows this in

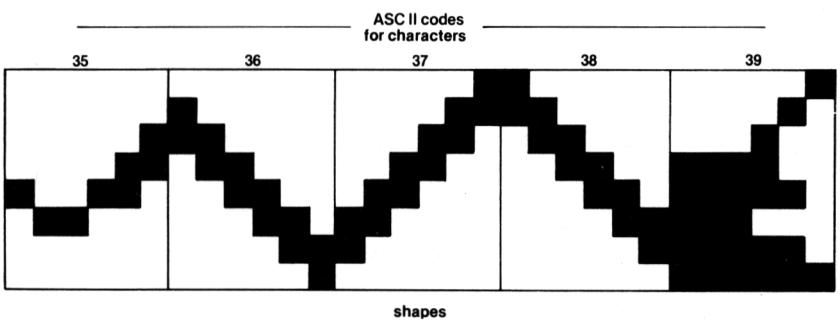


Fig. 6.7. A larger shape plotted in five grids. This can be printed as one string of characters.

action, and since you now know what it's about, I don't need to say much more. I have used the five character codes 35 to 39 inclusive, and allocated shapes to each. Because the whole 6 × 8 grid has been used, the parts of the 'worm' will join up when you see them printed.

```
10 CLS : PAPER 14: INK 2
20 GENPAT 0,39,4,8,16,240,248,224,248,2
52
30 GENPAT 0,38,128,192,96,48,24,12,4,0
40 GENPAT 0,37,4,12,24,48,96,192,128,0
50 GENPAT 0,36,0,128,192,96,48,24,12,4
60 GENPAT 0,35,0,0,4,12,152,96,0,0
70 CSR 5,10
80 LET WORM$="#$%&'"
90 PRINT WORM$
100 GOTO 100
```

Fig. 6.8. The worm printing program.

Printing is done by combining the characters into a string and printing the string. It's simple, hard work, and fascinating to look at!

More creations

In addition to using the normal ASCII codes of 32 to 127 for creating characters, you can use codes that are not part of the ordinary ASCII set. In particular, you can use GENPAT to create characters for codes 129 to 154. These codes normally produce either a blank or a ink-coloured block. The most useful of these are codes 129 to 143, because these are the codes for the programmable keys, F2 to F8, and SHIFT/ F1 to SHIFT/ F8. When you want to code numbers in this range, you must use mode number 1, so that each GENPAT line will start with GENPAT 1, rather than with GENPAT Ø, followed by the character number. The computer does not allow you to use code 128 for user-defined character, however, so F1 cannot be used. Using any of the other F keys will give the character shape when the key is pressed. The precise action depends on whether you are using the key as an immediate instruction or not. For example, if you make up a new shape for the key F2, and use a GENPAT line to form this shape, you will see the shape on the screen when you press the F2 key. If you then press RET, however, you will briefly see CLS appear, then the screen will clear! In other words, the key still keeps its action, even when it has been reprogrammed in this way. If you type a PRINT" instruction, and then press the F2 key, you will see the shape appear, and you can then type the other quote mark. When you press RET, however, the shape will change to the letters CLS! The *action* will be to print the

shape this time, however. This can appear *very* confusing in a program listing, so for programs that you want to pass to other people, it's probably better to keep to the code numbers. If you use PRINT CHR$(129), then this isn't altered in any way as the program is listed.

Graphics and screens

It's time now to turn away from the text screen, and to plunge into a completely new realm of graphics programming. Of all the many machines I have used, the Memotech is one of the easiest on which to create good graphics effects, but as always, you have to know the rules. The first rule is that you can have text screens, which can display text and user-defined characters, or graphics screens. The graphics screens can also display text, and they can use a wider range of colours and commands. You can also have the screen appearing split, part text, part graphics, because of the unique Memotech feature of *virtual screens*. Yes, that's a phrase that I'll have to explain.

A virtual screen means an area on the screen which acts like a miniature screen in its own right. You have, in fact, been working with this all the way through. The ordinary operating mode of the Memotech uses three of these virtual screens, one for listings, one for entry and one for messages. These act independently of each other, and don't get in each other's way. The valuable feature of the Memotech is that you can also use other shapes and sizes of screens that you define yourself! Before we get to that stage, let's see what is available as standard.

A virtual screen is selected by using the VS instruction, followed by a space and then a number (Ø to 7). Some of these numbers have been allocated already. When you switch on, you use virtual screen Ø, which is the entry part. Virtual screen 1 is the listing screen of 19 lines, and virtual screen 7 is the single-line message screen. While a program is running, all of the screen area is used, and this complete screen is virtual screen number 5. All of these are 'text screens', meaning that you place shapes on them by use of the PRINT instruction, and instructions like PAPER and INK will affect all the characters on a screen.

There is another member of this set – virtual screen 4. This is a graphics screen, the full size of your TV or monitor screen, and it's the one that we will use for graphics work. The PAPER and INK instructions will work on this screen, and PRINT can be used to position characters on the screen, but it can also be used for a

specialised set of drawing commands. This screen is selected by using VS 4, but whenever a program that has used this screen comes to an end, the normal set of text screens will return, awaiting your next instruction. So that we don't feel too adventurous, then, we'll start to investigate VS 4 with a few instructions that we know already.

The program in Fig. 6.9 shows a flavour of what is to come. VS 4

```
10 VS 4: CLS
20 PAPER 4: INK 11
30 PRINT "This is a message on the grap
hics screen"
40 PRINT : PRINT : PAPER 3: INK 6
50 PRINT "and so is this!"
60 GOTO 60
```

Fig. 6.9. Using the graphics screen by placing VS 4 in the program.

selects the graphics screen, and the CLS instruction must be used to clear this screen. That's important, because the ordinary RUN action, or typing CLS as a direct command, or using key F2, all clear the *text* screen, leaving the graphics screen untouched. If you want proof of this, try deleting the CLS from line 10 later on, and running the program several times in a row. Remember that you will have to use BRK to stop the program each time. Line 20 then selects PAPER and INK colours, and line 30 prints a message in these colours. In line 40, however, another set of PAPER and INK colours is chosen. Now, if you did this with the text screen, the new colours would replace the old ones, but the graphics screen acts differently. The first phrase remains in the colours that were chosen for it, and the rest of the screen uses the new colours! On the graphics screen, then, you are free to choose colours to a much greater extent. As we go on, you will find that there are also simpler commands for the purpose.

You will also notice a few other points. One is that the graphics screen, VS 4, does not take up the whole of your TV screen. It leaves a blue border round most of the screen. Early models of Memotech suffered from a displacement of this screen, so that the left side had no border, and some details at the extreme left side could not be seen. If your TV has a 'line-hold' control, you can sometimes adjust this to bring the left side into view. If your TV has no such control, and haven't any experience of TV adjustment, leave it alone, and design your programs so that they don't use the extreme left-hand side. It may well be that the problem has been cured by the time this book appears. Another thing that you will notice about the graphics

screen is the appearance of the text. The letters appear to be much more widely separated than they are on the text screen. This is not an illusion – they *are* separated further apart. The reason is that the graphics screen makes use of 'character' sizes which are drawn on an 8×8 grid. Now our letters are drawn on a 6×8 grid, so there are two more spaces placed between them when we use the graphics screen. Just to emphasise the point, the program of Fig. 6.10 will print each letter in a different colour. This is a good test of how well your TV is

```
10 VS 4: CLS : CSR 5,8
20 LET A$="Memotech again!"
30 FOR N=1 TO 15
40 INK N: PRINT A$(N,1);
50 NEXT
60 GOTO 60
```

Fig. 6.10. Printing letters in colour on the graphics screen.

able to display all the colours of the Memotech. Only a colour monitor will be able to display each colour really well, and a lot of domestic TV receivers will show some colours as very fuzzy. Of all the domestic TV receivers, the Sony Trinitron performs best in this test, mainly because the Memotech is set up to suit this particular model.

Plottings and drawings

The graphics screen can display text, and in colour as well, but it's main reason for existing is graphics. It's time, then to look at a few of the graphics commands that we can use. These are PLOT, LINE and CIRCLE. Each of these instructions has to be followed by numbers, and you have to know what the numbers mean, and how they are used. We'll start with PLOT.

PLOT is a word that we often use in connection with graphs, and its use with the Memotech is similar. PLOT means 'make a dot appear'. We call the dots *pixels*, short for *picture elements*, and when we use PLOT, we will make a pixel change from PAPER colour to INK colour, so that it becomes visible against the background. It will only become visible, of course, if you have chosen an INK colour which *is* visible. INK $\emptyset$ is *never* visible, and ink of the same colour as the PAPER colour can't, of course, be visible. PLOT needs two numbers following it, and there must, as usual, be a space

between the 'T' of PLOT and the first number. The first number is a left-to-right position, and it can take values of ∅ to 255. A value of zero means the extreme left-hand side of the screen, and a value of 255 means the extreme right-hand side. This number is often referred to as the *x co-ordinate*. The other number is the *y co-ordinate*, and its range is ∅ to 191. A value of zero means the *bottom* of the graphics screen, and a value of 191 means the top. Note that this numbering system is the opposite of the one that we use for the text screen, in which Y = ∅ means the *top* of the screen. These graphics numbers start at ∅, ∅ for the bottom left-hand corner no matter what size of screen we happen to be using.

An example will certainly help. Figure 6.11 shows PLOT being used to plot a graph. The machine plots graphs in the same way as we

```
10 VS 4: PAPER 6: INK 11: CLS
20 FOR X=1 TO 255
30 LET Y=80+80*SIN(2*PI*X/255)
40 PLOT X,Y
50 NEXT
60 GOTO 60
```

Fig. 6.11. Using PLOT to draw a graph, in this case a sine wave.

would – by placing a set of dots on the 'paper'. Line 1∅ sets things up, and the thing to note here is that CLS is placed *following* the PAPER command, so that the effect of CLS is to clear to the new paper colour. See for yourself what happens if you place CLS between VS 4 and PAPER 6! Line 2∅ then picks a range of values for X, and line 3∅ calculates a suitable value for Y. This value is obtained by finding the sine of the angle which X represents. A bit of adjustment is used to make the graph cover the whole screen in the correct way. The Memotech uses units of radians for measuring angle, and there are 2*PI radians in a complete turn (360 degrees). Since we have values of X from 1 to 255 available, we can make the range of 1 to 255 represent 2*PI radians. One unit of X therefore represents 2*PI/255 radians. The value of the sine of an angle varies between limits of −1 and +1, which wouldn't make much of a deflection in the Y direction, so we multiply by 8∅. Since this in itself would give negative values for Y, we then add 8∅ to make sure that this is impossible. We then end up with the expression in line 3∅. Line 4∅ plots these X and Y values to draw the graph. It's a simple, but extremely effective instruction that will be very useful if your interests are in graph drawing. Figure 6.12 shows how three graphs

```
10 VS 4: PAPER 6: CLS
20 FOR X=1 TO 255
30 LET Y=80+80*SIN(2*PI*X/255)
40 INK 11: PLOT X,Y
42 LET Y=80+80*(SIN(2*PI*X/255)^2)
44 INK 15: PLOT X,Y
46 LET Y=80+80*(SIN(2*PI*X/255)^3)
48 INK 4: PLOT X,Y
50 NEXT
60 GOTO 60
```

Fig. 6.12. Plotting three graphs. The colours interfere with each other when the lines are close together.

can be drawn in different colours. There is some interference between the colours, but you can see the separate traces quite satisfactorily.

The Memotech line

Plotting points is not exactly a fast or particularly satisfactory way of producing drawings, and the Memotech has a lot of other commands up its sleeve. One of these is LINE. LINE has to be followed by four numbers, separated by commas. As you should guess by now, there has to be a space between the 'E' of line and the first number. These numbers form two sets. The first pair are the X and Y numbers for the point at which the line starts. The second pair are the X and Y numbers for the point at which the line ends. The order is X, then Y, in each pair of numbers. It's a straight line, so these are the only points that we need. For example, if we used LINE 1Ø,1Ø,25Ø,19Ø this would draw a diagnonal line. The point X = 1Ø, Y = 1Ø is at the lower left-hand side of the screen, and the point X = 25Ø, Y = 19Ø is near the top right-hand side, so a line joining these points is a diagonal. Figure 6.13 is a plotting chart which you will find useful for planning drawings of this type, and also for the other graphics commands. You can make a full-sized copy for yourself by using graph paper that is scaled in centimetres and millimetres. Write down the numbers of 250 divisions across the long side of the graph paper, using each centimetre to represent ten divisions, and number from 0 to 180 upwards. The maximum X number is 255, and the maximum Y number is 191, but you don't need to use the extremes for most of your creations. If you then plot the points on

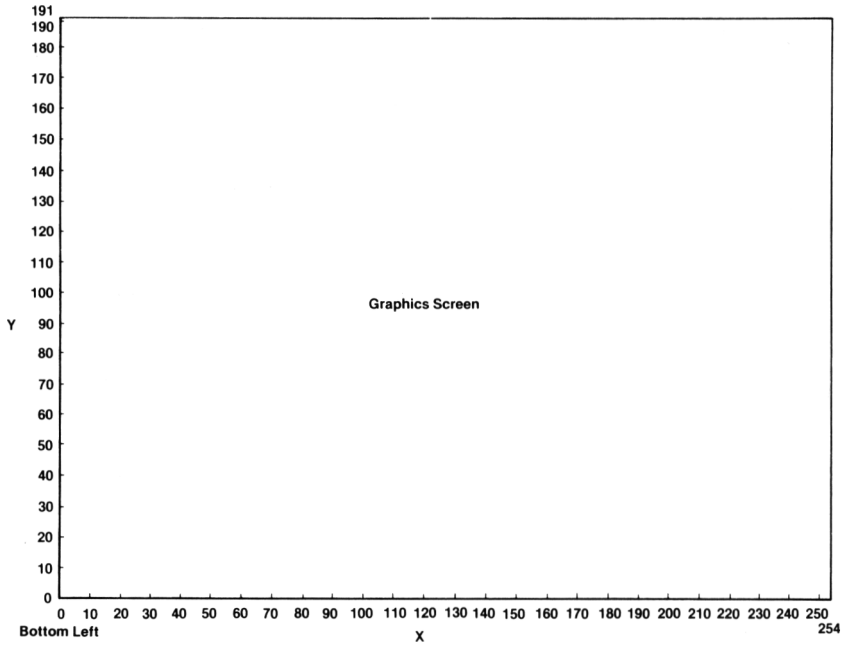


Fig. 6.13. A plotting chart for any graphics screen.

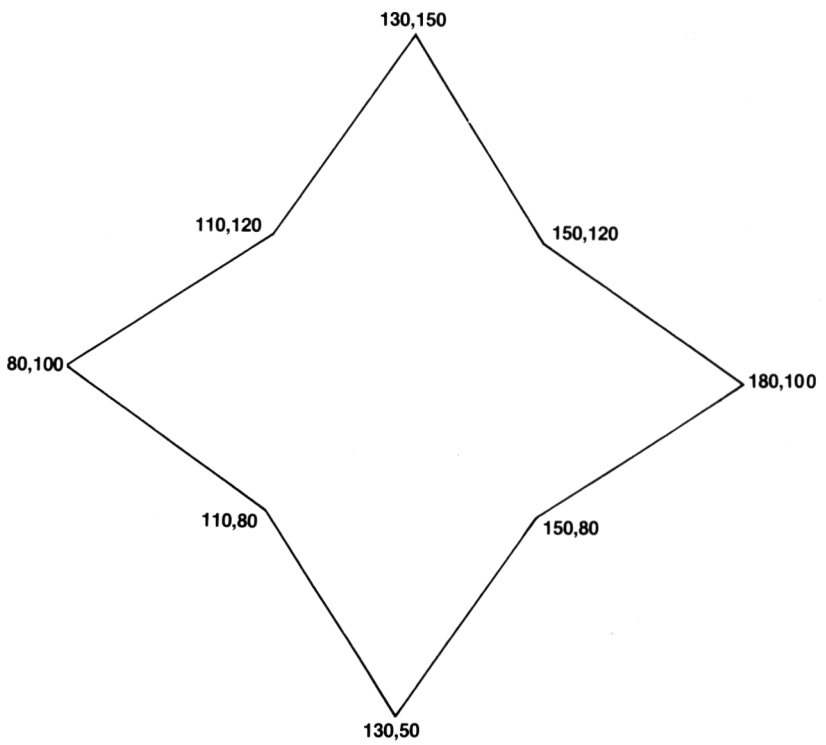


Fig. 6.14. A plan for drawing a star shape.

the graph as dots on the screen, you can plan your LINE commands much more easily. For example, Fig. 6.14 shows a star shape that has been planned on this graph. Each place where a line changes direction has been noted, and the X and Y numbers of these points are the numbers that we shall use in a program.

The next thing is how we program this. You could have a set of LINE instructions, one for each line that you want to draw. You would have to repeat the typing of all of the X and Y numbers, though, because the end point of one line is the start point of the next one. The Memotech does *not* allow you to omit the starting point of the LINE! The solution is shown in Fig. 6.15. We use a

```
10 VS 4: CLS
20 LET X=130: LET Y=150
30 FOR N=1 TO 8
40 READ X1,Y1: LINE X,Y,X1,Y1
50 LET X=X1: LET Y=Y1
60 NEXT
70 GOTO 70
100 DATA    150,120,180,100,150,80,130,
50,110,80,80,100,110,120,130,150
```

Fig. 6.15. The star program, using the LINE instruction.

READ...DATA instruction to read points. The variables X and Y are used for the starting point of a line, and X1 and Y1 are used for the end point. By using LET X = X1 and LET Y = Y1, we make the new starting point equal to the old end point on each pass through the loop. In this way, we need only read the *new* endpoint each time. It makes the use of LINE a lot simpler when you want to have a set of lines joined in this way. Having all of the data in one line, apart from the starting point, also makes the program much easier to edit.

LINE copes with the requirement for drawing straight lines, which is important for a lot of drawings. There is another instruction which will create circles. Reasonably enough, this is called CIRCLE, and it has to be followed by three numbers. These numbers are, in order, the X and Y numbers for the *centre* of the circle, and the size of the radius. In case you're a little rusty on circles, the radius is the distance from the centre to the rim of a circle. This radius is measured in the same units as X and Y, the size of the pixel dots on the screen. Figure 6.16 demonstrates this instruction, and draws a set of circles which have the same centre (concentric circles). Now these circles may not look very circular on a domestic TV. The problem is that unless a TV has been very accurately set up, circles

```
10 VS 4: CLS
20 FOR RADIUS=10 TO 80 STEP 10
30 CIRCLE 125,90,RADIUS
40 NEXT
50 GOTO 50
```

Fig. 6.16. Drawing circles, using CIRCLE.

never look very circular. A lot of TV sets are adjusted so that the picture width is too great. This is done deliberately, so that you don't notice that the width changes when the brightness of the picture changes. If your TV has a width control *on the outside of the cabinet* you can try adjusting it. Another way is to adjust the 'height' control, if there is one accessible. On a lot of modern TV receivers, however, these controls are inside the cabinet, and you should *not under any circumstances* meddle with anything inside the cabinet unless you are an experienced TV serviceman. In particular, you should never open a TV cabinet when the set is plugged in. If you must have perfectly circular circles, have an expert set up your TV, or use a monitor and see pictures of better quality than you ever thought possible!

With that off my chest, it's time to see how we can brighten up things a bit. The LINE and CIRCLE illustrations have used the *default* colours, blue background and white lines, with which you are issued when you switch on the Memotech. Up to now, we have used PAPER and INK to change these colours, but it's time now to meet another colour changing instruction. Appropriately enough, it's called COLOUR, and it can be used only when you are working with a graphics screen.

Chameleon Memotech

The COLOUR instruction has to be followed by two numbers. The second of these is quickly disposed of, it's the colour number, range Ø to 15, that we have used also for PAPER and INK. The first number is new, though. It's a mode number that decides what a COLOUR instruction will do. The table in Fig. 6.17 shows the use of this number, whose values can be Ø to 4. At first sight, you might wonder why we don't use just two numbers, one for PAPER and one for INK. The reason is that text and graphics need different treatment. For a text character, we have to colour each dot in the character. For a graphics point, we have to colour only one point. In

Mode No.	*Effect*
∅	Background for text (PAPER)
1	Character colour for text (INK)
2	Background for graphics.
3	Foreground for graphics.
4	Border around full screen.

Fig. 6.17. The use of the mode numbers for COLOUR.

some types of program, this would not make any noticeable difference. For example, take a look at the program in Fig. 6.18. This carries out a drawing twice, once using COLOUR ∅ and 1, and the next time using COLOUR 2 and 3 in the same drawing. You

```
10 VS 4
20 COLOUR 0,2: CLS : COLOUR 1,11
30 GOSUB 1000
40 PAUSE 5000
50 COLOUR 2,2: CLS : COLOUR 3,11
60 GOSUB 1000
70 GOTO 70
80 STOP
100 DATA  70,65,200,65,200,130,70,130
1000 LET X=70: LET Y=130
1010 FOR N=1 TO 4: READ X1,Y1
1020 LINE X,Y,X1,Y1: LET X=X1: LET Y=Y1
1030 NEXT : RESTORE 100
1040 LINE 85,65,90,50: LINE 90,50,95,65
1050 LINE 175,65,180,50: LINE 180,50,18
5,65
1060 CIRCLE 90,50,10: CIRCLE 180,50,10
1070 CSR 13,12: PRINT "BLUEBELL"
1080 RETURN
```

Fig. 6.18. Making a drawing with different COLOUR conditions.

might see these two drawings appear to be different if you switch on the computer, load this program, and then run it, but you will see no difference the next time you run it, if you haven't switched off or used NEW. That's because once COLOUR modes 2 and 3 have been allocated, they stay allocated until we change them. This means that these settings will exist when you run the program a second time, even though the line which sets the COLOUR has not been run. Because of this, colour commands for text only appear to have no

effect, and the LINE commands are controlled by the previous COLOUR settings. Note, incidentally, the instruction RESTORE 1ØØ in line 1Ø3Ø. Line 1Ø1Ø reads DATA, and when the last data item has been read, there are no more. RESTORE 1ØØ has the effect of starting any later READ at the beginning of line 1ØØ, where the DATA is. In this way, we can read the same data each time the subroutine is called. We could, if we liked, have more than one set of DATA in different lines, and cause READ to take different DATA each time.

Now take a look at Fig. 6.19, which illustrates when the COLOUR numbers *do* matter. The program starts by setting all of

```
10 VS 4: FOR N=0 TO 4: COLOUR N,1: NEXT
 : CLS
15 COLOUR 4,13: REM BORDER
20 COLOUR 0,6: CLS : COLOUR 1,4
30 LINE 150,5,150,190
40 COLOUR 1,10
50 LINE 5,80,250,80
60 CSR 2,10: PRINT "TEXT": PAUSE 8000
70 COLOUR 2,6: CLS
80 COLOUR 3,4
90 LINE 150,5,150,190
100 COLOUR 3,10
110 LINE 5,80,250,80
120 CSR 2,10: PRINT "PLOT"
130 GOTO 130
```

Fig. 6.19. The difference between text and graphics colouring.

the colours to COLOUR 1, which is black. This is necessary, because without this line, the program will behave differently on different runs. The colour settings are not changed by the command RUN, so that you get a false impression of what the program does the second time you run it unless these numbers are all changed. In line 15, COLOUR 4 is used. This sets the colour of the *border* of the main screen. The next lines, 2Ø to 6Ø appear to draw lines and text, but all that you will see is the text. This is because the colour settings are text colour settings, and they affect only the area of screen where this is text! The rest of the screen remains in the colours that were set at the start, black. Lines 7Ø to 12Ø then use the plotting colour commands, and you can see how different the effect is. The program has to be stopped by using BRK as usual, but this does not affect the values of the colour numbers. If you removed the loop in line 1Ø, you

would find that the program ran quite differently second time round, because the values that were allocated in lines 8Ø and 1ØØ would still affect the drawings in lines 3Ø and 5Ø! This can cause a lot of confusion if you are not aware of it.

Create-a-screen time

Up till now, we have used VS 4 for all of our graphics work. It's time now to expand our horizons, and look at how we can create our own screens. This makes use of the CRVS instruction, which has to be followed by seven numbers. This is not a simple instruction, and you need some practice before you can use it to its best effect. We'll start with an illustration, and then get down to the how-and-why department.

Figure 6.20 illustrates the creation of two virtual screens, one graphics screen and one text screen. As you will probably realise by

```
10 CRVS 2,1,16,0,15,5,32
20 VS 4: CLS : VS 2
25 COLOUR 2,6: COLOUR 3,11: COLOUR 4,15
30 CLS : LINE 2,2,100,36
40 LINE 2,36,100,2
45 PAUSE 8000
50 CRVS 3,0,0,14,12,9,40
60 VS 3: PAPER 15: INK 3: CLS
70 CSR 2,2: PRINT "This is a text scree
n."
```

Fig. 6.20. Creating screens with CRVS.

now, the two behave very differently, and this, combined with the action of virtual screens, can cause some odd effects until you understand what's going on. We'll start with the CRVS instruction in line 1Ø. The first number that follows CRVS is the virtual screen number. When you are creating your own screens, it's as well to avoid the screen numbers that are used for other purposes, so that you are left with the choice of 2, 3 or 6. If you choose any of the other numbers, Ø to 7, they will automatically return to normal at the end of a program, so that you need to use an endless loop to keep them visible. This is not always a disadvantage, because it's a simple and automatic way of ensuring that these screens are cleared. For our graphics screen, we are going to use 2.

The next number in the CRVS instruction of line 1Ø is 1, which makes this a graphics screen. The use of Ø would create a text screen. The next numbers, 16 and Ø, specify the position of the top left-hand corner of the screen. These are *not* the X and Y position numbers that you would use with LINE, PLOT or CIRCLE, but the numbers that you would use CSR. The number 16 specifies a position just over halfway along the screen width, and Ø specifies the top of the screen. The next two numbers specify the number of character position and number of lines for this screen. By using 15 characters, we take up most of the rest of the line, and the 5 allows us 5 lines. Finally, the number 32 specifies the 'normal' number of characters per line. This is 32 for a graphics screen (remember that the graphics characters are more widely separated than the text screen characters). You can achieve interesting effects by altering this number, but that's just a bit too advanced for the moment! Note that if anything goes wrong with these commands, you get the 'SE.B' error message, which the Manual states means 'Invalid ESC sequence'!

The effect of line 1Ø, then, is to define what the size and position of Virtual Screen 2 will be. Because it's a graphics screen, we can use commands like COLOUR, PLOT, LINE and CIRCLE. It will be surrounded by the ordinary graphics screen, VS 4, however. If we don't clear VS 4, anything that is on that screen will appear around VS 2. Line 2Ø attends to this, and starts the virtual screen 2. Line 25 then selects colours, and lines 3Ø and 4Ø place a cross in the small virtual screen. You have to be careful about the numbers that you use for LINE, PLOT and CIRCLE instructions. Each virtual *graphics* screen behaves as if it were independent. Its X and Y values for the bottom left-hand corner are always Ø,Ø, but the top right-hand values will depend on how large you have made the screen. For a graphics screen, the maximum X number is 8 times the width in characters, minus 1, and the maximum Y number is eight times the number of lines, minus 1. For a screen which is 15 characters across by 5 lines deep, we could use values of X up to (15*8)−1 = 119, and values of Y up to (5*8)−1 = 39. The PAUSE in line 45 gives you time to look at all this and admire your work.

After the PAUSE, things happen again. Line 5Ø creates another virtual screen, and this one is a text screen. It will be a VS 3, and the Ø following the 3 is what determines that it's a graphics screen. At this instant, the graphics screen vanishes. You *cannot* have a text screen and a graphics screen existing together. As well as erasing VS 2, line 5Ø also removes VS 4, and replaces it by the normal full text screen, VS 5. Once again, we use the numbers to settle the position and

dimensions of this screen. Position numbers Ø, 14 make the top left corner of the screen start at CSR position Ø, 14. The next two numbers fix the size of the screen as 12 characters across by 9 lines deep. The final figure is 4Ø, because this is a text screen. Since this is a text screen, the rest of the screen is also a text screen. The colours have to be controlled by using PAPER and INK, rather than COLOUR. In addition, PAPER and INK affect *all* of the text screens together; you can't have different colours in different text screens at the same time. A text screen is a much simpler and more restricted thing than a graphics screen.

Just to emphasise this point, take a look at the program of Fig. 6.21. This one uses VS 4, the normal graphics screen, and lays on to

```
10 VS 4: CLS
20 LINE 0,0,254,191
30 LINE 0,191,254,0
40 CRVS 2,1,10,0,10,5,32
50 VS 2: COLOUR 2,6: COLOUR 3,3
60 CLS : CIRCLE 39,19,15
70 CRVS 3,1,10,18,10,5,32
80 VS 3: COLOUR 2,10: COLOUR 3,13
90 CLS : LINE 0,19,79,19
100 LINE 39,0,39,39
110 GOTO 110
```

Fig. 6.21. Creating two graphics screens.

```
10 VS 4: CLS
20 LINE 0,0,254,191
30 LINE 0,191,254,0
40 CRVS 2,1,10,0,10,5,32
70 CRVS 3,1,10,18,10,5,32
80 VS 2: GOSUB 1000
90 VS 3: GOSUB 2000
100 PAUSE 1000
110 VS 2: GOSUB 2000
120 VS 3: GOSUB 1000
130 PAUSE 1000
140 GOTO 80
1000 COLOUR 2,6: COLOUR 3,3
1010 CLS : CIRCLE 39,19,15
1020 RETURN
2000 COLOUR 2,10: COLOUR 3,13
2010 CLS : LINE 0,19,79,19
2020 LINE 39,0,39,39
2030 RETURN
```

Fig. 6.22. Illustrating independent actions on two screens.

it two more graphics screens, 2 and 3. These two additional screens use different background and foreground colours, and have different patterns drawn on them. All three screens exist at the same time, and can be manipulated independently of each other. This is something that very few computers up to now have been able to do so simply, and it's another good reason for preferring the Memotech for graphics work. Just as an inkling of what is possible, take a look at the program of Fig. 6.22. This uses the same screens as the previous program, but swaps the small screens around, leaving the main screen constant. You can see from this how the screens can be controlled independently, and it should set you thinking as to how you can use this remarkable and spectacular technique. Just to keep your memory fresh, Fig. 6.23 summarises the rules for creating virtual screens.

(1) A text screen and a graphics screen cannot be used at the same time.
(2) Any PAPER and INK command will affect all text screens alike.
(3) Graphics commands such as PLOT cannot be used on a text screen. Only PRINT CHR$(X) can be used to place graphics shapes on the text screen.
(4) A text screen uses 40 character positions, of which 39 are normally used. A graphics screen uses 32 character positions.
(5) Text characters are plotted on a 6×8 matrix, graphics characters are plotted on an 8×8 matrix.
(6) Several graphics screens can exist at the same time, and text can be placed on them if wanted. The letters of this text will be spaced out by more than the spacing on the text screen.

Fig. 6.23. Summary of rules for using virtual screens.

Chapter Seven

Guide to Greater Graphics

ANGLE, PHI and line

The graphics instructions that we have looked at so far are very useful, and a lot of computers permit nothing further in the way of graphics. There are limits, however, to what you can achieve with just straight lines and circles. In addition, the method of plotting that uses X and Y numbers has some disadvantages. Suppose, for example, that you wanted to produce a square, and then draw a set of squares of different sizes. With X, Y graphics, you need to calculate a new set of X and Y numbers for each different square. There's another way of drawing patterns which gets around this limitation, and that's what we're going to look at now.

Suppose you defined a square this way. Take a point. Draw a line of 'K' points in length, upwards. Turn your direction of drawing through 90 degrees and repeat the line. Turn again through 90 degrees, and draw the line again. Do another turn, another line. That makes a square because all squares have four 90 degree angles. If you want a small square, make the value of 'K' small; if you want a large square, make the value of 'K' large. That's all there is to it. Here's another advantage. Suppose that you want to draw the square tilted. All you have to do is to draw the *first line* at a different angle. The rest of the lines follow the same pattern of four 90 degree angles. It's a different method of thinking about drawings. This system is sometimes called *polar co-ordinates*; the X-and-Y method is called *Cartesian co-ordinates*. We'll start work on polar co-ordinates by looking at three important instructions, ANGLE, PHI and DRAW.

Your turn?

The first of the new instructions is ANGLE, ANGLE has to be

followed by a number, which will be taken as an angle to the horizontal, expressed in radians. The purpose of the ANGLE instruction is to specify a starting direction for all the drawing that follows. If you want to take your starting direction as horizontal, left-to-right, for example, then ANGLE must be followed by a zero. For a lot of drawings, it doesn't matter very much what your starting direction is, so ANGLE $\emptyset$ is as good a start as any other.

You may not be familiar with angles measured in radians, however. This is no place to go into the why and how of measuring angles, but you need to know how to make use of radians. The best starting point is that a right angle, 90 degrees, is equal to PI/2 radians. The number that we call PI can never be written exactly, which is why we always refer to it as PI(π). It's the ratio of the distance round a circle (the circumference) to the distance across (the diameter), and it appears in the use of radians because a circle is the result of a turning action. The Memotech is equipped with PI – if you type PRINT PI, for example, you will get the result to eight places of decimals. The variable name of PI is reserved for this value, and you can't assign it to anything else. Figure 7.1 shows the values of some useful angles in terms of PI, and how you can convert from degrees to radian units, using PI. If you type ANGLE PI/2, then,

Angle in degrees	*Angle in terms of PI (π)*
30	PI/6
45	PI/4
60	PI/3
90	PI/2
180	PI
360	2*PI

In general, an angle of 'D' degrees will be PI/N, where N = 180/D.

Fig. 7.1. Familiar angles in terms of PI.

you set your starting direction for any drawings as vertically upwards on the screen. If you set ANGLE PI, then the direction is horizontal again, but directed right-to-left this time. ANGLE 3*PI/2 gives a starting direction that is vertically downwards. The change of ANGLE is *always anticlockwise*, which is why ANGLE PI/2 gives an upward direction, not a downward direction.

Now a starting direction isn't quite enough. We need a command which specifies a direction for the next line that we draw. This

command is PHI, and once again it has to be followed by an angle in units of radians. The angle that is used here is *added to the angle that we used in the ANGLE command.* For example, if we used ANGLE ∅ (horizontal, left to right), then PHI PI/2 would mean vertically upwards. It's vertically upwards, not vertically downwards because PHI, like ANGLE always operates in an *anticlockwise* direction. The difference between ANGLE and PHI, however, is that PHI values accumulate. What that means is that each value of PHI is added to the previous one. If you start with ANGLE ∅ (horizontal), and then have PHI PI/2, your direction for drawing is vertically upwards. If later in the program, you use PHI PI/2 *again*, then the direction of drawing turns through another right angle, anticlockwise. This makes the direction horizontal, right to left. Another PHI PI/2 will make the direction vertically *down*, and so on. There's no doubt about it, an example would make all of this a lot easier.

Figure 7.2 shows a simple program that draws a square. As a

```
10 VS 4: CLS
20 ANGLE 0
30 PLOT 75,45
40 PHI 0
50 DRAW 100
60 PHI PI/2
70 DRAW 100
80 PHI PI/2
90 DRAW 100
100 PHI PI/2
110 DRAW 100
120 GOTO 120
```

Fig. 7.2. Square bashing – Memotech style.

program, it's rather poor, because all of these repeated instructions should have been put into a subroutine, and called in a FOR...NEXT loop. I wanted to illustrate the uses of ANGLE, PHI and the other instruction, DRAW, however, and it's a lot easier to see in this form. We start with the usual VS 4 and CLS instructions to prepare the graphics screen. ANGLE ∅ gives the starting direction as horizontal, left to right, and PLOT 75,45 provides the starting point for drawing. Lines 4∅ and 5∅ then cause a line of 1∅∅ units (the units are of pixel size) to be drawn horizontally, left to right. The next step, in line 6∅, is to alter the direction of drawing through PI/2 (a right angle), so as to point vertically upwards. Another DRAW 1∅∅ then draws a line of 1∅∅ units length in this direction. Lines 8∅

and 9Ø then alter the direction to horizontal, right to left, and draw another line. Lines 1ØØ and 11Ø then provide the last line of the square, vertically down. Each DRAW instruction has to be followed by a length in pixel units, and will draw a straight line. This line will start where the previous line ended, or at a point which you specify by using PLOT. The direction of the line will be the direction that is specified by the combined value of PHI and ANGLE.

Suppose we make this into a subroutine, with a few improvements in technique. Figure 7.3 shows what can be done. Lines 1Ø to

```
10 VS 4: CLS
20 ANGLE 0
30 PLOT 10,10: PHI 0
40 FOR L=5 TO 180 STEP 5
50 FOR N=1 TO 4
60 GOSUB 1000
70 NEXT N
80 PAUSE 500
90 NEXT L
100 GOTO 100
1000 DRAW L: PHI PI/2
1010 RETURN
```

Fig. 7.3. Using a squares subroutine to make a 'corridor'.

3Ø set everything up, with ANGLE Ø and PHI Ø, and a starting point of 1Ø, 1Ø. We then use two loops. The outer loop, using L as its variable, is going to be used to draw squares of different sizes, and L will be the length of the side. We have to choose a maximum value of L which will not be outside the limits of the screen (X not more than 254, Y not more than 191). Line 5Ø starts an inner loop. This is the loop that actually draws the square by calling the draw-and-turn routine four times. Line 8Ø provides a PAUSE so that you can see what is happening, then the next value of length of side is used until all of the squares have been drawn. It's very impressive, and even more so if you remove the PAUSE step. Looks like one of those long corridors, doesn't it?

Turn again ...

It's time now to look at another advantage of this system of drawing. Because the value of ANGLE is always the starting direction for any drawing, altering ANGLE will produce drawings that are differently

tilted. Take a look at Fig. 7.4 now. Once again, this has been put into simple form rather than using a subroutine. Line 2Ø defines a set of different starting values for ANGLE, using a loop. Line 3Ø ensures that the same starting point is used for the drawings, and the rest of

```
10 VS 4: CLS
20 FOR R=0 TO 2*PI STEP .1: ANGLE R
30 PLOT 150,100
40 PHI PI/2
50 DRAW 50
60 PHI PI/2
70 DRAW 50
80 PHI PI/2
90 DRAW 50
100 PHI PI/2
110 DRAW 50
140 NEXT
150 GOTO 150
```

Fig. 7.4. Using ANGLE to rotate a drawing round its starting point.

the instructions from line 4Ø to line 11Ø simply draw a square. Because of the effect of varying ANGLE in the loop, however, each square is tilted compared to the previous one. The result is rather a pleasing pattern, one which is quite a lot more difficult to produce using X, Y graphics. You'll notice that the effect is what you would get by rotating a square around one corner. If you want to produce the effect of rotating a square around its centre, then you have to plunge into a bit of geometry. Figure 7.5 shows a program which will

```
10 VS 4: CLS
20 FOR R=0 TO 2*PI STEP .5: ANGLE R
30 PLOT 125+35.35*COS(R-PI/4),100+35.35
*SIN(R-PI/4)
40 PHI PI/2
50 DRAW 50
60 PHI PI/2
70 DRAW 50
80 PHI PI/2
90 DRAW 50
100 PHI PI/2
110 DRAW 50
140 NEXT
150 GOTO 150
```

Fig. 7.5. How to rotate a square around its centre.

carry out this action. Line 3ø is the one which ensures that the square turns around its centre, and Fig. 7.6. shows how I arrived at line 3ø. Graphics of this type are a lot easier if you know a bit about geometry, but don't let that put you off. Armed with a few formulae,

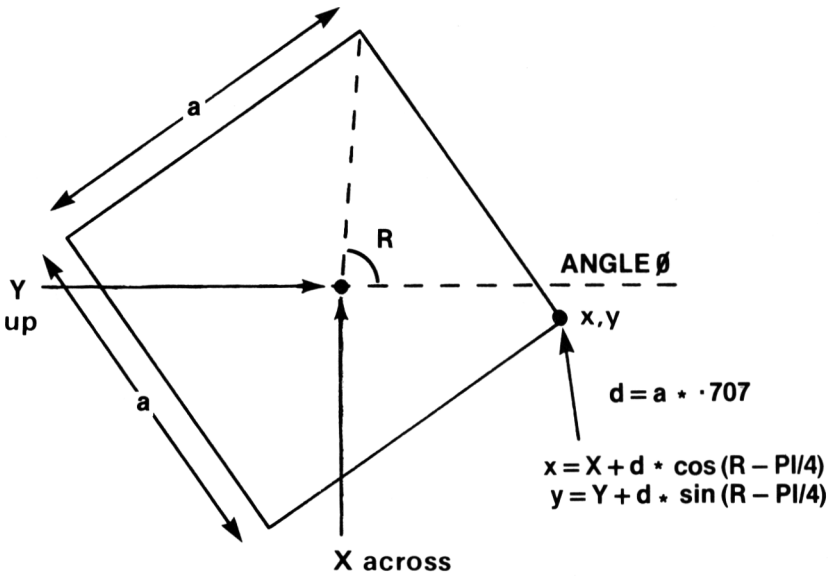


Fig. 7.6. For the geometrically strong only – how to rotate the square round its centre.

and with a bit of experience, you can produce results as good as the geometry whizz-kids. Knowing what you want to do is the main thing, you'll soon find out *how* with a bit of practice.

Arc de Triumphe?

So far, we're still stuck with straight line and circle drawings, even though we can now do rather more interesting things with them. Another instruction word, ARC, allows you to construct curves. Like many graphics instructions, it needs some grasp of geometry to get the best from it. That's why this set of graphics instructions is used so much now for teaching geometry by means of the LOGO language.

ARC has to be followed by two numbers. Of these the first one is a distance. It's the distance measured along the curve of the arc, and that's where the difficulties start. We normally measure distances in

straight lines, you see, and a measurement along an arc is not the same. If you draw an arc, and join its ends with a straight line (Fig. 7.7), you can see that the distance measured along the arc is *always* greater. The second number in the ARC instruction is an angle in the usual radian units. This is the angle through which your drawing direction changes when you draw the arc.

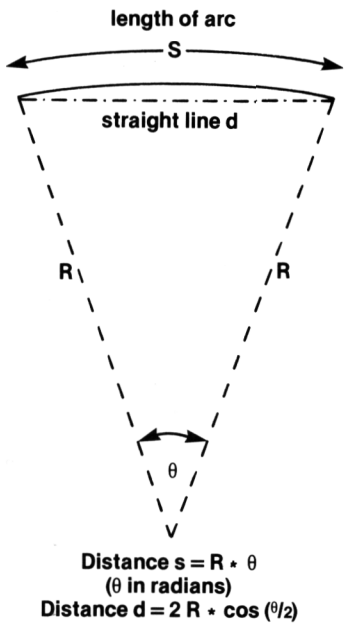
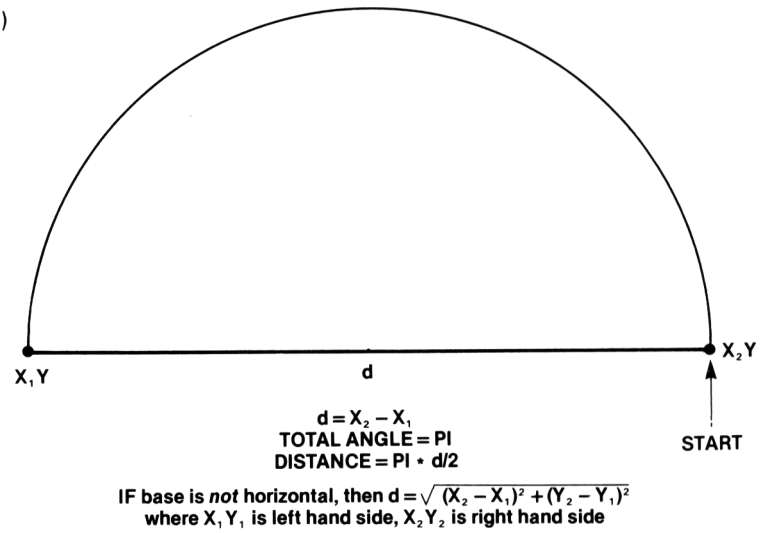


Fig. 7.7. Length and angle for an arc.

The simplest illustration of the use of ARC is in drawing a semicircle. Figure 7.8 illustrates this in drawing (a) and in program (b), starting with a straight line which is produced by the familiar LINE command. Now at this point we have to be careful. The LINE command draws a line from the point 2Ø,2Ø to the point 2ØØ,2Ø – a horizontal line. The ARC instruction will *not*, however, take the point 2ØØ,2Ø as its starting point. You have to use a PLOT to specify this starting point. If you don't realise this (and it's not exactly stressed in the manual!), then you can waste a lot of time on attempts to produce ARC drawings. Line 4Ø in Fig. 7.8(b) therefore sets up ANGLE as zero, the starting point as 2ØØ,2Ø, and the starting angle (PHI) as PI/2. Now why do we use PI/2? The reason is another piece of geometry. The outside of a circle is *always* at right angles to a diameter drawn to the same point. If we want to draw a semicircle starting at 2Ø,2ØØ, we have to aim in a direction at 90 degrees, which

(a)



(b)

```
10 VS 4
20 COLOUR 2,2: COLOUR 3,5: CLS
30 LINE 20,20,200,20
40 ANGLE 0: PHI PI/2: PLOT 200,20
50 ARC 282.74,PI
60 GOTO 60
```

Fig. 7.8. Drawing a semicircle.

is PI/2 radians. That's hurdle number one. The next one is, how long is a semicircle? The answer is PI*D/2, where D is the diameter. In the program, this has been put into figures (because I planned the program before the computer was available!), but since the diameter is 18ø (it starts at X = 2ø and ends at X = 2øø), you could use a distance of 18ø*PI/2 in the ARC instruction. Once again, if you don't know the geometry, you can still learn to use the formulae.

Figure 7.9 shows something rather more ambitious, a 'barrel' or TV-screen shape constructed from four arcs. The cunning part of this program is the calculation of PHI and ARC values. As a program, it's quite a simple one which uses a subroutine to repeat the arc drawings. Let's see how the values come about, though. Lines 1ø and 2ø set up for the drawing, with a starting point of 15ø,6ø. The first awkward bit is the value of PHI, which is 1.32. What we're going to do is to draw an arc of 5ø units length, and we want to aim it so that the end of the arc will be vertically above its start. Figure 7.10 shows the calculation that produces the value of 1.32 for this, and I have written it down as a formula so that you can use it for yourself.

```
10 VS 4: CLS
20 ANGLE 0: PHI 1.32: PLOT 150,60
30 ARC 50,.5
40 FOR N=1 TO 3
50 GOSUB 1000
60 NEXT
120 GOTO 120
1000 PHI 1.06: ARC 50,.5
1010 RETURN
```

Fig. 7.9. Drawing a complicated arc shape.

What you have to do is to pick a *centre of curvature* for the arc. This will decide how curved the arc is. The distance from the centre of curvature to the line of the arc is called the *radius of curvature*. A large value of radius of curvature will give you an arc which looks almost like a straight line. A small value of radius of curvature will produce a very curved arc, and if you make the radius of curvature too short you may find that the arc cannot be drawn at all!

Having launched the shape in line 3Ø of Fig. 7.9, you have to produce three more arcs to close the shape again. This is done by using the subroutine, but a different value of PHI angle is needed in this subroutine. This is because the end of one arc is directed at an angle which is not vertical or horizontal – it depends, in fact, on the values of distance and angle in the previous ARC instruction. Once again, this is a piece of geometry, and Fig. 7.10 shows the method so

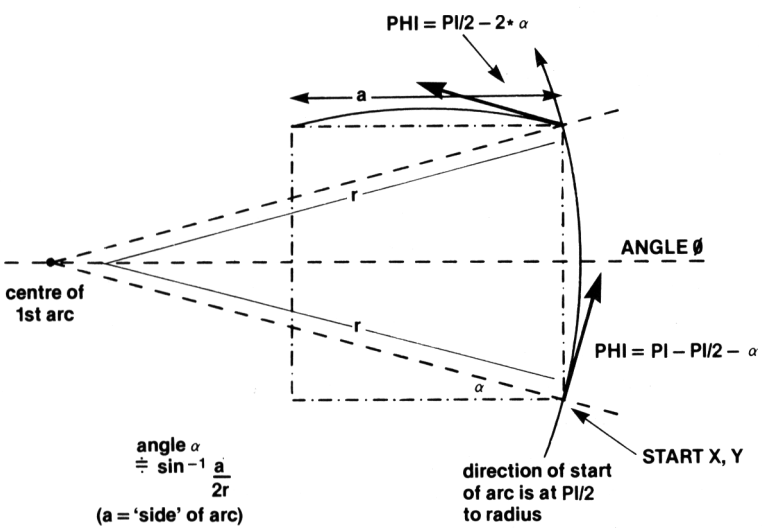


Fig. 7.10. Calculations behind the arc shapes – for geometry students only!

that you can make use of it for yourself. Older readers may wish that they had paid more attention to geometry at school, younger readers may wish that geometry was taught at their school. One thing is certain – keep plugging away at these graphics, and you'll know a lot more geometry than anyone else in your street!

Lay it on the line

Before we continue to explore the graphics abilities of the Memotech, another few words on planning may be useful. The more elaborate your graphics become, the more planning they need, but the process of planning is not necessarily more difficult. The essential part is to be able to locate points on your plan, and so it's very important to make your plans either on graph paper, or on tracing paper clipped over graph paper. No matter which type of graphics you use, you will always have to plot points, and when you plan your drawings on graph paper, it's much easier to see the shape and scale of the result than if you just 'think of a number'. Just to rub it in, Fig. 7.11 shows a much more elaborate graphics plan. This was

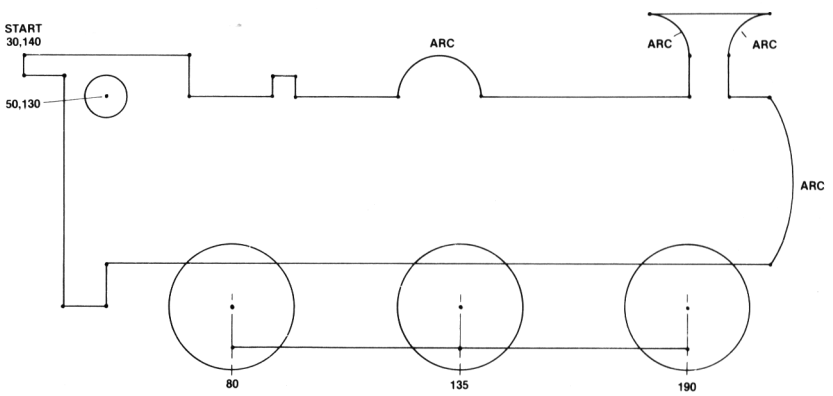


Fig. 7.11. A graphics plan outline.

originally drawn on graph paper, so that the X and Y values of each point could be read from the graph scales. The plan shows a mixture of straight lines, circles and arcs, so that both types of graphics commands are being used. The main point to remember when you do this, is that you will usually need a PLOT command before you start to use ARC or DRAW instructions.

Figure 7.12 shows the program which resulted from the plan in Fig. 7.11. The endless loop which holds the picture on the screen has

```
10 VS 4
20 COLOUR 2,2: COLOUR 3,1: CLS
30 LET X=30: LET Y=140
50 FOR N=1 TO 9
60 READ X1,Y1
70 LINE X,Y,X1,Y1
80 LET X=X1: LET Y=Y1: NEXT
90 LINE 190,140,190,130
100 LINE 190,130,140,130
110 LET X=120: LET Y=130
120 FOR N=1 TO 8
130 READ X1,Y1
140 LINE X,Y,X1,Y1
150 LET X=X1: LET Y=Y1: NEXT
160 DATA  30,135,40,135,40,80,50,80,50,
90,210,90,210,130,200,130,200,140
170 DATA  120,130,95,130,95,135,90,135,
90,130,70,130,70,140,30,140
180 CIRCLE 50,130,5
190 CIRCLE 80,80,20
200 CIRCLE 135,80,20
210 CIRCLE 190,80,20
220 PLOT 140,130: ANGLE 0: PHI PI/2
230 ARC 10*PI,PI
240 PLOT 190,140: ANGLE PI/2: PHI 0
250 ARC 5*PI,PI/2
260 PLOT 210,150: ANGLE PI: PHI 0
270 ARC 5*PI,PI/2
280 LINE 180,150,210,150
290 LINE 80,70,190,70
500 GOTO 500
```

Fig. 7.12. The Loco drawing program.

been placed in line 5ØØ so that you can, if you like, add a few more lines of detail (what about the buffers, before I get expelled from the Steam Loco League). The main point, however, is to convince you that graphics like this need planning, and to illustrate how this planning is carried out. Unplanned graphics can be fun, but they can also waste a lot of time and patience.

Attributes - mainly physical

Just in case you thought by this time that you had all of the graphics

commands of the Memotech under control, there's yet another command word, ATTR. ATTR is a shortened version of *attributes*, and it refers to the way the computer behaves when the graphics screen is used. We have used this idea already, because COLOUR is a form of attribute control. When you type COLOUR 3,5, then you are making each graphics point that is plotted or drawn be of colour number 5. You could say that this was a colour attribute. The attributes that are controlled by the ATTR command are a lot more subtle, and you will need some experience before you can use them correctly. Well, let's face it, that's what this book is for.

The ATTR command needs to have two numbers following it. The first number decides what is to be controlled. The second number switches the control on and off. The second number can be ∅ or 1. A ∅ will switch the attribute off, and a 1 will switch the attribute on. Once an attribute has been switched on, then it stays on until it is switched off. While it is switched on, it will affect each of the commands that causes anything to appear on the screen. To work, then.

Figure 7.13 demonstrates ATTR ∅,1 in action. ATTR ∅ affects characters that have been placed on the graphics screen with a PRINT instruction, and the effect of ATTR ∅,1 is to reverse the characters. That doesn't mean reverse their shapes, but reverse their

```
10 VS 4
20 COLOUR 0,13: COLOUR 1,10: CLS
30 CSR 0,4: PRINT "THIS IS NORMAL PRINT
ING"
40 PRINT : PRINT
50 ATTR 0,1
60 PRINT "THIS IS WITH ATTR 0,1"
70 PRINT : PRINT
80 ATTR 0,0
90 PRINT "BACK TO NORMAL AGAIN"
100 FOR J=1 TO 50
110 ATTR 0,1: GOSUB 1000
120 PAUSE 200: ATTR 0,0
130 GOSUB 1000: PAUSE 200
140 NEXT
150 GOTO 150
1000 CSR 4,15: PRINT "FLASHING LETTERS!
"
1010 RETURN
```

Fig. 7.13. Using ATTR ∅.

colours. A character is normally printed in INK colour on PAPER background, and for the graphics screen we use COLOUR 1 and COLOUR Ø for these two respectively. After ATTR Ø,1, the colours are reversed, as Fig. 7.13 shows. ATTR Ø,Ø restores things to normal, and if you program a loop, you can make a set of letters flash, as lines 1ØØ to 14Ø illustrate. It's a useful way of making titles stand out, and yet another reason for using the graphics screen in place of the text screen.

The next attribute is ATTR 1. As usual, this is switched on by ATTR 1,1 and off by ATTR 1,Ø. Like ATTR Ø, it is for use with characters that have been printed on to the graphics screen. This time, however, the effect is to overprint. Overprinting means that a character printed on top of another one does *not* replace the first one but adds to it! Figure 7.14 gives you a taste of the effect. It's particularly useful for adding the accents and other marks that foreign languages need, and it's a special boon for languages teachers.

```
10 VS 4
20 COLOUR 0,14: COLOUR 1,5: CLS
30 CSR 5,3: PRINT "1000"
40 CSR 2,6: PRINT "Now for slashed zero
s"
50 PAUSE 2000
60 CSR 5,3: ATTR 1,1
70 PRINT " ///"
80 CSR 2,8: PRINT "Now for Spanish spel
ling."
90 CSR 5,10: PRINT "El canon."
100 ATTR 1,1: CSR 10,10: PRINT "~"
110 ATTR 1,0: CSR 2,15: PRINT "O.K.?"
300 GOTO 300
```

Fig. 7.14. Using ATTR 1.

The next two attributes are intended to affect the graphics, as distinct from the character-printing, commands on the graphics screen. ATTR 2 is an 'unplot' attribute. If you have used ATTR 2,1 in a program, then each PLOT, DRAW, LINE, CIRCLE, ARC, etc., that you use from then onwards will be in background colour instead of in foreground colour. If you have a subroutine which draws a shape, then drawing it with ATTR 2,Ø will cause the shape to appear, and drawing it with ATTR 2,1 will make it disappear! Figure 7.15 illustrates this using a diamond shape. The loop that

```
10 VS 4
20 COLOUR 2,12: COLOUR 3,5: CLS
30 FOR N=1 TO 50
40 ATTR 2,0: GOSUB 1000
50 PAUSE 500
60 ATTR 2,1: GOSUB 1000
70 PAUSE 500: NEXT
100 GOTO 100
1000 LINE 140,20,180,80
1010 LINE 180,80,140,160
1020 LINE 140,160,100,80
1030 LINE 100,80,140,20
1040 RETURN
```

Fig. 7.15. Drawing and undrawing with ATTR 2.

starts in line 3Ø will draw and then 'undraw' the diamond pattern whose shape is determined by lines 1ØØØ to 1Ø3Ø. When ATTR 2,Ø is used, the shape is drawn, but when ATTR 2,1 is used, the shape is undrawn, and the lines are rubbed out. This can also be used to make a pattern on top of another one. Figure 7.16 shows a blue rectangle which is created by using a loop to draw a set of lines in foreground colour. By using ATTR 2,1, the program then draws a diamond

```
10 VS 4
20 COLOUR 2,11: COLOUR 3,4: CLS
30 FOR Y=30 TO 150
40 LINE 20,Y,200,Y
50 NEXT
60 ATTR 2,1
70 LINE 110,150,20,90
80 LINE 20,90,110,30
90 LINE 110,30,200,90
100 LINE 200,90,110,150
110 ATTR 2,0
300 GOTO 300
```

Fig. 7.16. Changing colours with ATTR.

shape in the background colour. Using ATTR in this way avoids having to change colours with COLOUR.

The last of the ATTR commands is ATTR 3. When you switch this on by using ATTR 3,1, then anything that is plotted in foreground colour is changed to background colour, and anything in background colour is changed to foreground colour. This can cause a lot of interesting effects, as the program of Fig. 7.17 shows. This sets up a rectangular shape in foreground colour, and then draws a strip

```
10 VS 4
20 COLOUR 2,6: COLOUR 3,11: CLS
30 FOR X=20 TO 200
40 LINE X,20,X,150
50 NEXT : GOSUB 500
60 LET AT=0: FOR P=30 TO 150 STEP 30
70 GOSUB 1000
80 NEXT
90 ATTR 3,0
100 GOTO 100
500 ATTR 2,1: FOR Y=60 TO 70
510 LINE 20,Y,200,Y
520 NEXT : ATTR 2,0: RETURN
1000 IF AT=0 THEN  LET AT=1 ELSE  LET A
T=0
1005 ATTR 3,AT
1010 FOR Q=50 TO 80
1020 LINE P,Q,P+40,Q
1030 NEXT
1040 RETURN
```

Fig. 7.17. Using ATTR 3.

across it in background colour. Some rectangles are then drawn, using ATTR 3 alternately switched on and then switched off. With ATTR 3,1, you can see the foreground and background colours reversing, but with ATTR 3,Ø in operation, the drawing operation of subroutine 1ØØØ produces a rectangle in foreground colour alone.

That isn't all, because these attributes can be combined, although this is something that you won't do very often. Nevertheless, there can be some benefits from combining attributes, as Fig. 7.18

```
10 VS 4
20 COLOUR 2,8: COLOUR 3,5: COLOUR 4,1:
CLS
30 ATTR 2,1: ATTR 3,1
35 CSR 4,5: PRINT "BOTH ATTRIBUTES ON"
40 GOSUB 1000
60 ATTR 2,0: ATTR 3,0
65 CSR 4,5: PRINT "BOTH ATTRIBUTES OFF"
70 GOSUB 1000
100 GOTO 100
1000 FOR Y=20 TO 170
1010 LINE 20,Y,200,Y
1020 NEXT : RETURN
```

Fig. 7.18. Combining ATTR instructions.

demonstrates. With attributes 2 and 3 both on, the effect is – nothing! Any PLOT, LINE, DRAW or other graphics instruction has no visible effect. This can be useful if you want to move a plotting point invisibly from one place to another. One very important point about the use of attributes is that you should always end a program with all attributes reset to zero. If you end a program with some attributes set and operating, then this can affect the next program that you run. Nothing short of switching off will have much effect, and you may be totally baffled as to why you can't clear a screen, for example. When you encounter problems of this type, take a close look at your attributes.

Spy on the screen

Before we move on to even more remarkable graphics effects, there's another instruction, GR$, which can be useful. The main use of this instruction is for allowing you to print graphics patterns on paper, if you have a graphics printer like the Epson, but it has another use which is only briefly mentioned in the manual. This other use is detecting what is present at any position on the screen, and we can illustrate its use by the old-time 'bouncing-ball' routine. The principle is a simple one. We create two 'walls' which are lines drawn in foreground colour. A point is made to move by alternately plotting it, then unplotting it, moving it slightly and repeating the process. When the walls are straight and fixed, we could make the 'bounce' happen simply by reversing the direction of movement of the point whenever it reached the correct value of X or Y. A better method, however, is to detect the wall because it is in foreground colour. By using this method, you don't have to know the values for X or Y when the point reaches the wall.

Figure 7.19 shows the Memotech version of this program. Lines 1Ø to 6Ø simply set up a black screen with two vertical yellow lines. These are the walls. Line 7Ø sets values of X and Y that correspond to a point at the top centre of the screen. The variable K is a 'direction variable'. When K = 1, then the movement of the point is left-to-right. When K = −1, then the movement is right-to-left. This happens because the point is moved by adding K to X.

Line 8Ø calls the print-and-move subroutine. This plots a point at the position given by the values of X and Y, then unplots it using ATTR 2,1. The attribute is then switched off again, and line 1Ø2Ø shifts the plotting position. While K is positive, the direction of

```
10 VS 4
20 COLOUR 2,1: COLOUR 3,11: COLOUR 4,1:
 CLS
30 FOR Y=1 TO 191
40 LINE 5,Y,6,Y
50 LINE 250,Y,251,Y
60 NEXT
70 LET X=122: LET Y=190: LET K=1
80 GOSUB 1000
90 IF GR$(X+K,Y,1)=CHR$(1) THEN  LET K=
-K: LET Y=Y-1
100 IF Y>2 THEN  GOTO 80
300 GOTO 300
1000 PLOT X,Y
1010 ATTR 2,1: PLOT X,Y: ATTR 2,0
1020 LET X=X+K
1030 RETURN
```

Fig. 7.19. A 'bouncing-ball' program, using GR$.

movement is to the right. When the subroutine returns, the *next* position for the point is tested. GR$(X, Y, 1) tests one single point at position X, Y, and the result of GR$ is a *character*. This means you cannot equate it to a number, and if it corresponds to a character which would have an ASCII code of less than 32, then it will not be possible to print it. If the point is at background colour, then GR$ will give the character whose ASCII code is zero. If the point is at foreground colour, then GR$ will give the character whose ASCII code is 1. Neither of these is a printable code, and there are two ways that we can detect them. One is by using the test:

IF GR$(X,Y,1) = CHR$(1)

and the other is by using the test:

IF ASC(GR$(X,Y,1) = 1

The important point is that you *cannot* use:

IF VAL (GR$(X,Y,1))=1 *or* IF GR$(X,Y,1)="1".

This syntax is not explained in the manual, and it is not at all obvious until you concentrate on the word 'character' in the description of the action!

When a wall is located just ahead of the point (because we are using X+K in GR$), then the value of K is inverted by using LET K = −K. If K was 1, it becomes −1, if it was −1 it becomes 1. The value

of Y is then reduced by 1, so that the point 'falls'. It's not exactly a lifelike bounce, but it's principles I'm aiming at, not polished perfection. The next step now is full scale animation, and that calls for the Memotech sprite commands. Next chapter please!

Chapter Eight
Identified Flying Objects

From what you have seen so far, animation of a shape on the screen can be a tedious process. It's bad enough to have to animate a point, and animating a shape with the ordinary commands of your average computer is like hacking salt from the Siberian mines, but with less pleasure attached. Nevertheless, this is about all that the average computer can do with BASIC language commands. Fortunately, we're not dealing with a computer that is 'average' in any respect. The Memotech has the ability to control moving objects, called *sprites*. You can determine the shape, and to some extent the size, of these sprites for yourself. They are controlled by BASIC instruction words, which makes the Memotech practically unique, for most computers use number codes to control sprites, and this makes it very hard to remember what you have to do. Memotech sprites can be created and controlled, once you have some practice in the art, without the need to keep the manual in one hand all the time!

Sprite creation

Working with sprites means that you have to determine the shape and size of the sprites, and then all that has to be done to move them. Keeping to the policy of one thing at a time, we'll start this chapter by looking at how a sprite is created. As we go on, you'll become more familiar with the ideas of sprites, and you will be able to take on the job for yourself, using your own ideas. As it happens, we have introduced some of the principles already, and all that you need at this stage is a bit of revision!

To create a sprite shape, we use the GENPAT instruction, which you met in Chapter Six. You'll remember that this consists of ten numbers, and the last eight of the numbers are the ones which decide the shape. The pattern is drawn on to an 8×8 grid, the code number

for each line worked out, and the set of eight numbers read from top to bottom of the grid. That's the revision part of it. However, the GENPAT instruction for a sprite is slightly different from the GENPAT for a character. The first difference is the first number that follows GENPAT. For character creation, we used $\emptyset$ or 1, and ignored 2 which allows you to create characters with different colours in each line. The simplest sprite creating number is 3, which allows you to create an 8×8 grid sprite. The second number in GENPAT is a 'pattern number', which is the equivalent of an ASCII code for that sprite. Each sprite must have its pattern number, just as we would use a different ASCII code for each user-defined character. For the moment, we'll stick to working with pattern number 127.

What we need now is an illustration, and Fig. 8.1 shows the first step, the 8×8 grid drawing and the GENPAT numbers. Note that

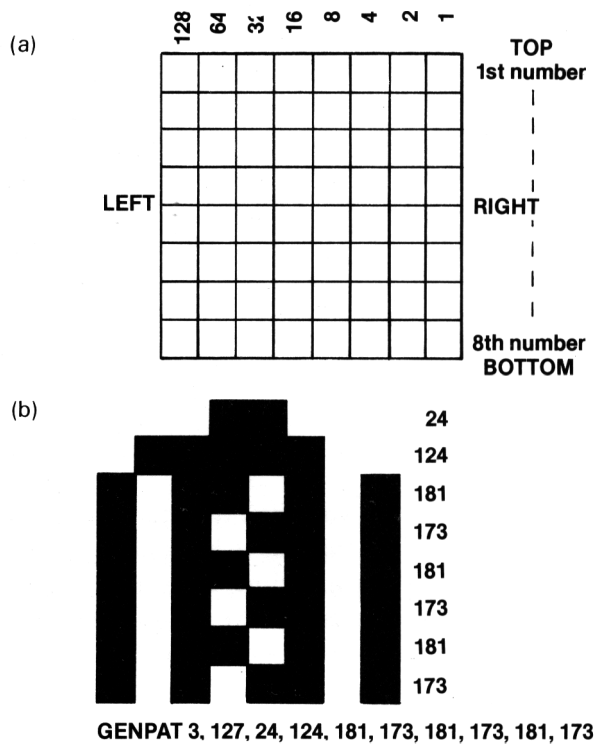


Fig. 8.1. (a) The 8×8 sprite planning grid, and (b) a shape planned on it.

this is an 8×8 grid, not the 6×8 grid that was used for user-defined characters. This shape is a monster from the deep, MFTD to you, and we can place the shape into the memory of the computer by

using the GENPAT instruction that is illustrated. This, however, doesn't make the shape appear on the screen. This is dealt with by another instruction word, SPRITE. SPRITE has to be followed by seven numbers. The first of these is the reference number for your sprite, which decides its importance. These reference numbers, or sprite numbers, can range from Ø to 31, giving you a total of 32 sprites in all. Each sprite must keep its own number because, as we shall see, this is how you can keep control over your sprites. The second number is the 'pattern number' which can be Ø to 127 for small sprites, or Ø to 31 for the large variety. In this example, we're using a small sprite, whose pattern number is 127.

So far, so good. Now we have to add the numbers that actually make the sprite appear. The next two numbers in SPRITE are X and Y position numbers. They refer to the screen positions in the usual way, but there is something distinctly unusual about them. Their maximum values are not the 254 and 191 that should be familiar to you by now, but −4095 to +4095. You won't see a sprite normally if you pick numbers outside the normal range of X and Y, but they will be waiting in the wings! For the example, we'll make X equal to 13Ø and Y equal to 85. The next two numbers control speed, one for the X direction and one for the Y direction. The speed number can take values from −128 to +127. A positive value of X means movement left to right, and a positive value of Y means movement upwards. Negative values simply mean movement in the opposite direction. A value of Ø means no speed, so no movement. The final number in the SPRITE command is a colour number, range Ø to 15, so that you can control the colour of your sprite.

We're not quite there yet. Before you can display a sprite, you have to give notice of how many sprites you want to control. This is done by using one of the CTLSPR (ConTrol SPRite) commands. The one we need is CTLSPR 2, which specifies how many sprites we will use. It's the equivalent of DIM for arrays, and no sprite ever gets displayed until this command has been carried out. Figure 8.2 shows where we've reached so far. Line 15 contains the vital CTLSPR 2,1

```
10 VS 4: CLS
15 CTLSPR 2,1
20 GENPAT 3,127,24,124,181,173,181,173,
181,173
30 SPRITE 1,127,130,85,0,0,11
300 GOTO 300
```

Fig. 8.2. First steps in sprite creation.

which tells the operating system that we're going to work with just one sprite. The GENPAT in line 2Ø then creates the sprite shape, and allocates 127 as its pattern number. This is the highest pattern number that we can use for a single-size sprite. Note that you will *not* see the shape appear when you type PRINT CHR$(127), because this is a sprite pattern number that we are using, not an ASCII code number. Line 3Ø then contains the SPRITE command. This is sprite number 1, pattern 127, and we want it placed at X = 13Ø, Y = 85. We want it to stay put, so the movement numbers are both zero. Finally, its colour number is 11.

Frankenstein's fun

Now that we have created this object, what about using it. The first thing to try is to move it. We'll start the object off at one corner of the screen and move it from there. Figure 8.3 shows the program modified so as to do this. To start with, we have to change the

```
10 VS 4: CLS
15 CTLSPR 2,1
16 CTLSPR 0,1
17 CTLSPR 5,1
20 GENPAT 3,127,24,124,181,173,181,173,
181,173
30 SPRITE 1,127,5,5,1,1,11
300 GOTO 300
```

Fig. 8.3. Animating a sprite. Animation by this method is a once-and-for-all process - you have no control over the sprite once it has been started.

SPRITE instruction in line 3Ø. We start at X = 5, Y = 5 now, and we have also added the speed number 1 (for X) and 1 (for Y). This by itself does *not*, however, make the sprite move. To do so we have to add two CTLSPR instructions. These *must* be placed ahead of the SPRITE command, or you will get an error message when the program reaches the SPRITE command. The first added one, in line 16, is CLTSPR Ø,1. This means that we have now decided to have 1 *moving* sprite. A moving sprite needs more attention from the machine than a non-moving one, so we have to give notice of our intentions. Line 17 then contains CTLSPR 5,1. The figure 5 selects the speed control action, and we have used speed 1, which is the fastest. With these modifications, we can see the sprite move at last. It moves slowly diagonally across the screen until it vanishes from sight.

Now is the time to play with this simple program. To start with, try the effect of different speed numbers in the SPRITE command. Using 1Ø,1Ø in this place certainly makes the movement brisker. Now go back to 1,1 as before, and try changing line 16 to CTLSPR Ø,5Ø. This, as you can see (if you wait long enough), has the opposite effect, and this value combined with the small numbers in the SPRITE command makes this good for snail racing. Now, what happens if you use CTLSPR Ø,5Ø along with higher numbers in the SPRITE command. Try it, you'll see that the movement is rather more noticeable this time, but in jerky steps. For the smoothest movement, you need to use large values in the SPRITE instruction, and a small number in CLTSPR Ø. You may wonder why there should be two ways of controlling speed. It's done, in fact, because this way you have much more control. You see, the CTLSPR commands affect all of the sprites, but a SPRITE command is specific to one sprite, which is why the SPRITE command carries the sprite number, and the CTLSPR command doesn't. When you are using a large number of sprites (any number more than one) then CTLSPR controls all of them, and you can speed them all up or slow them all down. By changing the SPRITE numbers, however, you can make one sprite move faster or slower than another, shift position, change colour, whatever you like. We'll see shortly that there is an easier way of making changes than running through the whole SPRITE set of numbers.

It's background that counts

One of the features of sprites is that they have a system of priorities. Priority means that you will always see a sprite pass across the screen even when other graphics are present. It appears, in other words, to move in front of the other graphics. Take a look at the program in Fig. 8.4. This creates a green background, and then draws some red columns of random length. A blue arrow then flies across, and because this is a sprite, it appears to pass in front of the columns so that you can see it all the way. Note, however, that you do *not* see the sprite in the border region of the screen. As always with colour graphics, the effect is very much better on a colour monitor than on a TV receiver.

Now what about one sprite passing over another? We can illustrate this with a few additions to the program, as illustrated in Fig. 8.5. Start by editing line 12Ø so that it is numbered 14Ø, and then

```
10 VS 4
20 COLOUR 2,2: COLOUR 3,13: COLOUR 4,1:
 CLS
30 FOR J=1 TO 5
40 FOR Y=0 TO INT(RND*190)
50 LINE 45*J,Y,45*J+5,Y
60 NEXT : NEXT
70 GENPAT 3,127,24,12,142,127,127,142,1
2,24
90 CTLSPR 2,1
100 CTLSPR 0,5
110 CTLSPR 5,1
120 SPRITE 1,127,0,50,25,0,5
300 GOTO 300
```

Fig. 8.4. Illustrating sprite priority over the background.

alter line 9ϕ and add lines 12ϕ and 13ϕ. The CTLSPR 2,2 now provides for two sprites. Only one of them will move, however, so we do not have to change the CTLSPR 5,1 in line 11ϕ. The new GENPAT makes this a sprite of pattern number 126, and its SPRITE numbers in line 13ϕ place it in the path of the arrow. It is, however, a non-moving sprite, and is coloured yellow. When you run this program, you will see the blue arrow, SPRITE number 1, pass in front of the yellow sprite, SPRITE number 2. This is another reason for numbering the sprites. The numbers are also a pecking

```
10 VS 4
20 COLOUR 2,2: COLOUR 3,13: COLOUR 4,1:
 CLS
30 FOR J=1 TO 5
40 FOR Y=0 TO INT(RND*190)
50 LINE 45*J,Y,45*J+5,Y
60 NEXT : NEXT
70 GENPAT 3,127,24,12,142,127,127,142,1
2,24
90 CTLSPR 2,2
100 CTLSPR 0,5
110 CTLSPR 5,1
120 GENPAT 3,126,16,16,255,255,255,255,
16,16
130 SPRITE 2,126,160,50,0,0,11
140 SPRITE 1,127,0,50,25,0,5
300 GOTO 300
```

Fig. 8.5. Priority of one sprite over another.

order. A low-number sprite will always appear to pass in front of a higher numbered sprite, and all sprites pass in front of the other graphics. With 32 possible sprites to play with, this opens up a lot of possibilities!

Still using the same program, we can look at another alteration. Add the line:

```
1Ø5 CTLSPR 6,1
```

to the program of Fig. 8.5. When you run it this time, you will see the sprites appear doubled in size! Since this is a CTLSPR instruction, it affects all of the sprites. Double size sprites are much more suited to a TV receiver, and you should try to work with this size unless you have a monitor, or particularly want to use the smaller size of sprites.

Bigger and better

Sprite sizes are not confined to the 8 × 8 grid that we have worked

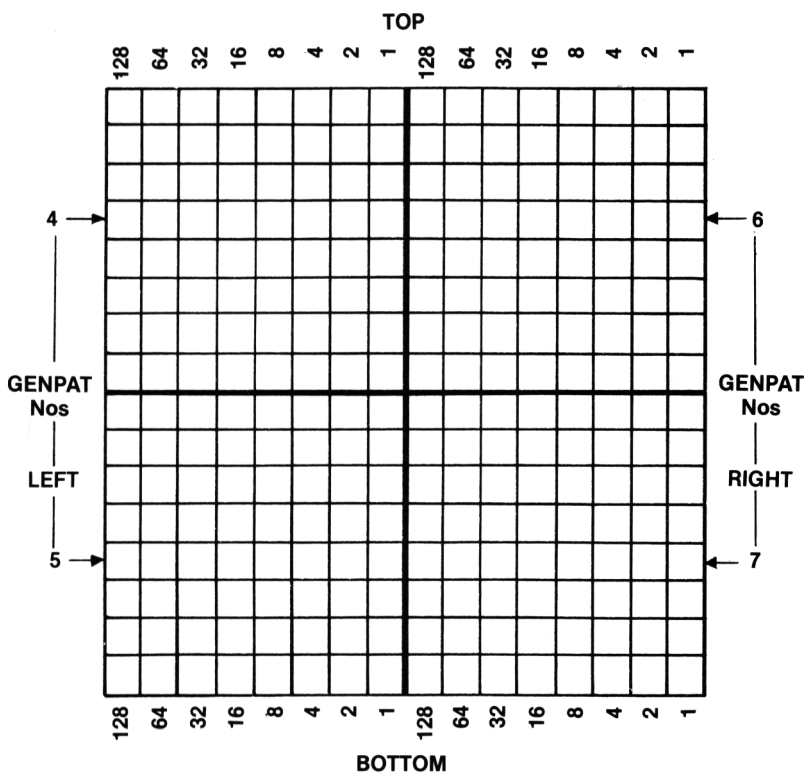


Fig. 8.6. A planning grid for large (16 × 16) sprites.

with so far. The Memotech provides for mammoth sprites (spritephants?) which are planned on a 16 × 16 grid. This consists of four 8 × 8 grids arranged as shown in Fig. 8.6., and it provides a new challenge. How do we design sprites on this grid?

To start with, we have to draw the shape on the grid. Figure 8.7 illustrates how this is done. As far as the drawing goes, this is much

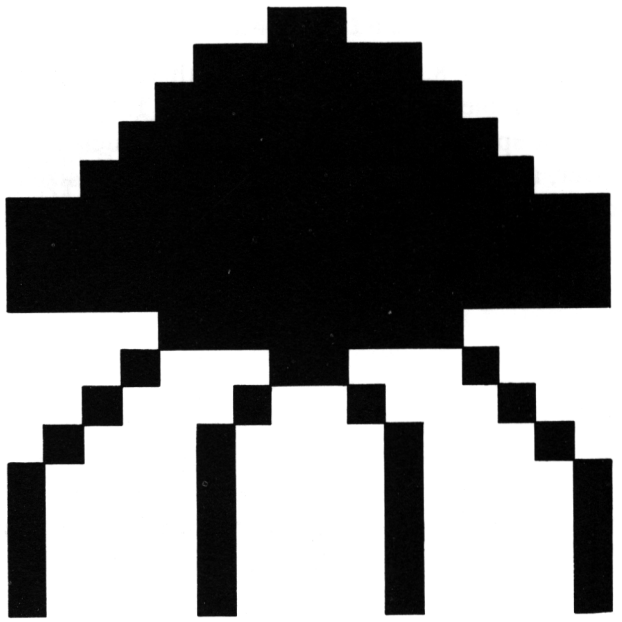


Fig. 8.7. A shape planned on the large grid.

the same as before, but with more squares to shade. When you come to decide the GENPAT patterns, however, there are four sets to work out. You treat each block of 8 × 8 squares as a separate GENPAT, and use the numbers which are shown alongside the blocks in Fig. 8.6. The top left-hand block uses GENPAT 4, the bottom left-hand block uses GENPAT 5, and so on. When you create a giant sprite, each of the four GENPAT lines must use the same pattern number, and it has to be in the range ∅ to 31 (not up to 127 as we could use for single size sprites). Furthermore, you will get an error message unless you prepare the computer for four GENPAT lines with the same pattern number! You must use CTLSPR 6,2 in some program line which comes earlier than the GENPAT. In the example (Fig. 8.8) I have put all the CTLSPR lines together before the GENPAT lines.

Looking at the example in more detail now, it starts with the usual

```
10 VS 4: CLS
20 COLOUR 2,1: COLOUR 3,12: COLOUR 4,1
30 CLS : ANGLE PI: PHI -.157
40 PLOT 254,0: ARC 254,.314
50 LINE 0,0,254,0
51 CTLSPR 0,5
52 CTLSPR 2,1
53 CTLSPR 6,2
54 CTLSPR 5,1
60 GENPAT 4,0,1,7,15,31,63,255,255,255
70 GENPAT 5,0,15,17,34,68,132,132,132,1
32
80 GENPAT 6,0,128,224,240,248,252,255,2
55,255
90 GENPAT 7,0,224,136,68,34,33,33,33,33
140 SPRITE 1,0,127,20,0,20,5
500 GOTO 500
```

Fig. 8.8. Animating the large sprite.

allocation of colours, and then uses the ARC instruction in line 4Ø to draw part of the surface of a very small planet. The CTLSPR lines 51 to 54 then specify a speed of 5, one sprite used, 16 × 16 size, and one sprite moving. The GENPAT lines create the pattern of a space excursion module, and line 14Ø animates it with the SPRITE command. Try it, and see the escape from Planet X in living colour! Having tried it, alter line 53 to read: CTLSPR 6,3, and then run the program again. This value of CTLSPR 6 will double the size of the 16 × 16 sprite, to give you the largest sprite that you can control as one unit. You can make even larger sprites if you are prepared to join several sprites together, because if you specify the correct starting places in the SPRITE instruction, and have them move at the same speed, then they will stay together. The CTLSPR instructions, remember, will affect all of the sprites equally.

Make an adjustment

As we have used them so far, the sprites move in the pattern that is set for them, and they are unaffected by the rest of the program. You set the sprites going with the SPRITE instruction, and while they move, you can be doing something else on the screen, like drawing lines or printing text. The computer is not really doing two things at once, it actually interrupts whatever else it happens to be doing to

attend to the sprites at intervals. It's because of this change at intervals action that the movement of the sprites can appear slightly jerky.

Now, we don't always want to have sprites operating independently of the rest of the program, though it certainly makes life easier for some purposes. There are ways in which we can change the sprite action at various times, however. One of these ways is the ADJSPR instruction. As you might guess, the command word stands for ADJust SPRite, and it has to be followed by three numbers. The first of these numbers decides what feature of the sprite you want to change. You can change the pattern number, so turning your sprite into another one. It's useful for changing toads into princes, or the other way round. You can also change the colour, X position, Y position, X speed or Y speed. You can't, however, change more than one of them in one ADJSPR instruction. The second number is the number of the sprite that you want to work on, and the third number is the value of the change that you want to make. If you wanted to change the pattern number from 127 to 126 for sprite 1, for example, you would use ADJSPR Ø,1,126. The Ø specifies a pattern change, the 1 specifies sprite number 1, and the 126 is the new pattern number. You must, of course, have a GENPAT line which has allocated a shape for pattern number 126. Figure 8.9 is a reminder of how ADJSPR is used. As an

ADJSPR mode no.	*Adjusts*	*Range of values*
Ø	pattern	Ø to 31 (large), Ø to 127 (small)
1	colour	Ø to 15
2	starting X value	Ø to 255
3	starting Y value	Ø to 255
4	X speed	Ø to 127 (right), 128 to 255 (left)
5	Y speed	Ø to 127 (up), 128 to 255 (down)

Fig. 8.9. How the ADJSPR instruction is used.

example of the use of ADJSPR, take a look at Fig. 8.10. This is the same space module program as we have used to date, but with a dramatic change! Line 15Ø provides a delay, and this is something that has to be worked out by trial and error. At the end of the PAUSE, the ADJSPR lines then cause the sprite to veer to the left and to change colour (any science fiction fan will tell you about that, it's Doppler shift!). There's another sprite adjustment instruction,

```
10 VS 4: CLS
20 COLOUR 2,1: COLOUR 3,12: COLOUR 4,1
30 CLS : ANGLE PI: PHI -.157
40 PLOT 254,0: ARC 254,.314
50 LINE 0,0,254,0
51 CTLSPR 0,5
52 CTLSPR 2,1
53 CTLSPR 6,2
54 CTLSPR 5,1
60 GENPAT 4,0,1,7,15,31,63,255,255,255
70 GENPAT 5,0,15,17,34,68,132,132,132,1
32
80 GENPAT 6,0,128,224,240,248,252,255,2
55,255
90 GENPAT 7,0,224,136,68,34,33,33,33,33
140 SPRITE 1,0,127,20,0,20,5
150 PAUSE 5000
160 ADJSPR 4,1,130
170 ADJSPR 1,1,7
500 GOTO 500
```

Fig. 8.10. The ADJSPR instruction in action.

MVSPR, which allows several adjustments to be made at the same time, but it's much quicker to use ADJSPR lines. We'll look at some uses for MVSPR shortly, but some ways of using MVSPR are so complicated that even the manual doesn't explain them in enough detail to make them usable!

Into orbit!

The sprite commands include provision for making a sprite appear to 'circle' around. Circling means that you will see the sprite appear to be travelling, perhaps, left to right across the screen. Normally, that's it, but when you specify a circling sprite, it will reappear, travelling in the same direction, at intervals. It's like getting a glimpse of a circling goldfish every time it passes your side of the bowl. This orbitting action is set by using CTLSPR 3. Following the CTLSPR 3 we need the number of orbitting sprites, and for our example, 1 is enough. Figure 8.11 shows another example, still based on the space module, so you only have to alter a few lines. The early part of the program is as it was, but following the PAUSE, things change. Line 16Ø adjusts the SPRITE speed number to

```
10 VS 4: CLS
20 COLOUR 2,1: COLOUR 3,12: COLOUR 4,1
30 CLS : ANGLE PI: PHI -.157
40 PLOT 254,0: ARC 254,.314
50 LINE 0,0,254,0
51 CTLSPR 0,5
52 CTLSPR 2,1
53 CTLSPR 6,2
54 CTLSPR 5,1
60 GENPAT 4,0,1,7,15,31,63,255,255,255
70 GENPAT 5,0,15,17,34,68,132,132,132,1
32
80 GENPAT 6,0,128,224,240,248,252,255,2
55,255
90 GENPAT 7,0,224,136,68,34,33,33,33,33
140 SPRITE 1,0,127,20,0,20,5
150 PAUSE 5000
160 ADJSPR 4,1,128
170 ADJSPR 1,1,7
180 ADJSPR 5,1,0
190 CTLSPR 3,1
200 CTLSPR 0,1
500 GOTO 500
```

Fig. 8.11. Creating an 'orbitting' sprite.

maximum in the leftward direction. Note, incidentally, how numbers of Ø to 127 are used in the ADJSPR command for speeds in the rightward direction, and numbers of 12Ø to 255 to mean leftward. The fastest speeds are 1 and 128 respectively. Line 17Ø changes the colour as before, and line 18Ø changes the Y speed to zero. This ensures that the sprite stops rising, so that its movement is purely horizontal. Line 19Ø then uses CTLSPR to specify orbitting, and line 2ØØ sets the fastest possible speed. Watch it zip along, but don't be too surprised at how long it takes to orbit. Remember that for the purposes of driving sprites, you are working on a 'map' which has 4096 points in each direction. When a sprite goes off the edge of the screen, it's having to move $2 \times 4096 - 256$ points, which is 7936 points. This takes time – if the sprite moves across the screen (of 256 points horizontally) in one quarter of a second, then it will take $.25 \times 7936/256$ seconds to orbit back to the other side of the screen, and that's 7.75 seconds. This, however, is another of these commands that you can set up and then leave to run by itself.

Free-range sprites

Sprites which carry out their pre-set action are, as we have seen, useful. Every now and again, though you want to keep a sprite completely under the control of a program. As you might expect by now, the Memotech provides for this as well. One way of providing for programming a sprite is to omit the CTLSPR command which allows movement. If you remove the CLTSPR 5,1 line, then you have no moving sprites. In place of this command, you can substitute CTLSPR 1,1Ø. The '1' part of this specifies that this is a distance command, and the 1Ø is the number of pixels. What this means is that it permits the sprite to be moved by a distance of ten pixels each time it is commanded. Next step – how is the movement commanded? Figure 8.12 shows the answer – a FOR...NEXT loop.

```
10 VS 4: CLS
20 COLOUR 2,1: COLOUR 3,12: COLOUR 4,1
30 CLS : ANGLE PI: PHI -.157
40 PLOT 254,0: ARC 254,.314
50 LINE 0,0,254,0
52 CTLSPR 2,1
53 CTLSPR 6,2
60 GENPAT 4,0,1,7,15,31,63,255,255,255
70 GENPAT 5,0,15,17,34,68,132,132,132,1
32
80 GENPAT 6,0,128,224,240,248,252,255,2
55,255
90 GENPAT 7,0,224,136,68,34,33,33,33,33
140 SPRITE 1,0,127,20,0,20,5
150 CTLSPR 1,1
160 FOR N=1 TO 260
170 MVSPR 1,1,INT(8-RND*4)
180 NEXT
500 GOTO 500
```

Fig. 8.12. Controlling a sprite more closely, using a loop.

In line 15Ø, the CTLSPR command makes sure that one pixel will be moved on each MVSPR command, because this is what is needed. The loop starts in line 16Ø, and extends to a range that should take the object off the screen. The key to it all is the MVSPR in line 17Ø. To move a sprite, we have to use MVSPR 1, and this has to be followed by the number of the sprite (Sprite number 1 in this example) and the direction code. The direction codes are as shown

on page 121 of the manual (provisional copy); they range from ∅ to 7, with 8 counting the same as ∅. This means that you haven't a very close control of precise direction, but it's ample for many purposes. By using INT (8 − RND*4) as the final number, we will make the direction change at random on each pass through the loop. It gives a splendidly out-of-control-spacecraft effect! Every now and again, the program causes a 'SE.B' error message, the reason for which is not exactly clear. The 'SE.B' message means 'Invalid ESC sequence', but it appears when the direction of the spacecraft causes an illegal number of MVSPR commands. Reducing the number in the loop will remove the message, but will cause the craft to appear stationary on the screen at the end of the loop. If you alter line 16∅ so as to read:

FOR N = 1 TO 2∅∅

then the sprite will not go out of bounds, and adding:

19∅ ADJSPR 1,1,∅

will make it disappear when the loop is ended. The reason for this is that we have changed the colour of the sprite to colour ∅, which is transparent! When it becomes transparent, then it disappears as far as we are concerned.

There's another way of controlling a 'free-range' sprite. This is by forcing the sprite to appear wherever we plot a point. This is not always convenient, because we don't always want to leave a trail of points following a sprite, but remember that you can plot with no trace! By combining ATTR 2 with ATTR 3, you can plot without

```
10 VS 4: CLS
20 COLOUR 2,4: COLOUR 4,4: COLOUR 3,11:
 CLS
30 CTLSPR 2,1
40 CTLSPR 4,1
50 CTLSPR 6,1
60 GENPAT 3,127,153,90,108,231,231,108,
90,153
65 SPRITE 1,127,0,0,0,0,2
70 ATTR 2,1: ATTR 3,1
80 FOR X=0 TO 254
90 PLOT X,80+80*(SIN(X*PI/128))^3
100 NEXT
110 ATTR 2,0: ATTR 3,0
500 GOTO 500
```

Fig. 8.13. Using a PLOT sprite, with ATTR used to suppress the trace.

```
10 VS 4: CLS
20 COLOUR 2,12: COLOUR 4,12: COLOUR 3,5
30 CLS
40 CTLSPR 2,3: CTLSPR 6,1
50 GENPAT 3,125,24,24,24,255,255,24,24,
24
60 GENPAT 3,126,129,66,36,24,24,36,66,1
29
70 GENPAT 3,127,255,129,129,129,129,129
,129,255
80 SPRITE 1,125,127,95,0,0,3
90 SPRITE 2,126,0,95,0,0,7
100 SPRITE 3,127,256,95,0,0,13
110 GOSUB 1000
120 VIEW 0,127
130 GOSUB 1000
140 VIEW 4,127
150 GOSUB 1000
160 VIEW 4,127
500 GOTO 500
1000 PAUSE 1000
1010 RETURN
```

Fig. 8.14. Using VIEW with sprites – the limitations.

making any point show on the screen, and so move a sprite without any trail behind it! An example is obviously required, so take a look at Fig. 8.13. The steps of CTLSPR and GENPAT should be familiar to you by now, but there is a new one – CTLSPR 4,1. This defines one PLOT sprite, a sprite which will be printed wherever a point is plotted. In every other respect this is a sprite pattern, so it does not have to be wiped out when the plot position is shifted. If we use line 7Ø to prevent the actual appearance of plotted points, then the sprite follows its mysterious path with no apparent reason. If you want to see the path plotted, then remove line 7Ø and try again.

A view from the window

There is so much that can be done using sprite graphics, moving or stationary, self-propelled or plotted, that you might think there could be no more. There is more, but to keep this book from expanding to the size of an encyclopaedia, we'll look at just one more sprite command. This is the VIEW command, which is unique

to Memotech. We have seen several times already how the sprites appear to move over a 'map' that is much larger than the screen. We can, if we like, imagine that the screen is a window looking on to this larger expanse. The screen size is 256 × 192 pixels, but the sprite *field* is 4096 pixels in each direction (up, down, left or right). The VIEW command allows you to shift the screen *window*, so that it looks at another part of the sprite field. Shifting the window by means of VIEW is a lot faster than moving a sprite, so it allows you to look at a number of sprites in sequence even when they are a long way apart. You can keep sprites stationary, just off the screen, and then suddenly display them by using VIEW. We can illustrate that use fairly easily, and also a warning that things are not always what they seem. Figure 8.14 shows a program which sets up three sprites. One is placed at the centre, and the other two at the edges. The sequence that starts in line 11Ø then shifts the viewing area. At first, we look at sprite number 1, then shifting the viewing area to the right places the sprite that was at the right-hand side into the centre. Reversing this then restores the original sprite to the centre, and the right-hand side sprite to the right-hand side. There is, however, no trace to be seen of

```
10 VS 4: CLS
20 COLOUR 2,12: COLOUR 4,12: COLOUR 3,5
30 CLS
40 CTLSPR 2,3: CTLSPR 6,1
45 CTLSPR 3,3
46 CTLSPR 0,10
47 CTLSPR 5,3
50 GENPAT 3,125,24,24,24,255,255,24,24,
24
60 GENPAT 3,126,129,66,36,24,24,36,66,1
29
70 GENPAT 3,127,255,129,129,129,129,129
,129,255
80 SPRITE 1,125,127,95,0,50,3
90 SPRITE 2,126,127,190,0,-20,7
100 SPRITE 3,127,127,-1,0,10,13
170 LET K$=INKEY$: IF K$="" THEN  GOTO
170
180 VIEW 6,20
200 GOTO 170
300 REM
500 GOTO 500
```

Fig. 8.15. Using VIEW with orbitting sprites.

the left-hand sprite! This action is not mentioned in the manual, and the effect seems to be that if the window moves too far away from a sprite, the sprite is lost! No amount of movement of the VIEW window will then let you see that sprite again. The values of −4095 and +4095 that are quoted in the manual for placing sprites in the SPRITE command do not seem to apply to this use, but only to the starting position for an *orbitting* sprite. You can, however, use VIEW to catch a glimpse of orbitting sprites, and Fig. 8.15 shows an example, using the same shapes as before. This sets three sprites into orbit, and allows you to tap any letter key, or the space-bar, to find the sprites again. Try to hold the yellow cross in the screen area, and then see if you can find when two of the shapes coincide. There's the basis of a good 'fruit-machine' type of game in this!

Chapter Nine
Textual Topics

We seem to have spent a lot of time on graphics, mainly because the graphics capabilities of the Memotech are so outstanding. This chapter now tries to redress the balance by dealing with some more of the equally excellent text commands of the Memotech, and in particular with the NODDY text processing language. As always, we shall start in a simple and straightforward way, and then pile on the complications. Unlike previous chapters, though, this one cannot be so generously illustrated with listings, because the NODDY language does not lend itself so well to listing.

We'll start by tying up a few loose ends, commands that we didn't deal with earlier. The first two of these are DSI and SPK$. DSI is a command which is a form of loop that incorporates an INKEY$ action. When the DSI instruction is found in a program, the screen clears, and is completely blank. When you press a key, however, a letter appears, and it looks as if you can use the screen for typing in the usual way. Most of the keys operate normally, but RET stops the process. Figure 9.1 gives you a taste of this, along with the use of another rather useful command, SPK$. SPK$ means 'the character at the cursor'. When you use this command in a loop, the character

```
10 DSI
20 CSR 0,0
30 FOR N=1 TO 50
40 LET A$(N)=SPK$
50 NEXT
```

Fig. 9.1. Using SPK$ and DSI for text.

that is obtained by means of SPK$ can be assigned to a string variable, and the action of SPK$ includes incrementing the cursor position. When you use the program in Fig. 9.1, type a line of words, and then press the RET key. Lines 3Ø to 5Ø will then place the characters that you have typed into a string variable. A$. If you then

type PRINT A$, you will see your characters reproduced. Unlike INPUT, it is not put off by the presence of commas. You must place the cursor where you expect to find the first character, however, and if you press the down-arrow key while you are typing, you may put some characters in a place where they cannot be read by SPK$, because they are more than 50 character places from the starting position. You may find that the lack of a cursor is a disadvantage, so try the version in Fig. 9.2. The cursor is restored by using PRINT

```
10 PRINT CHR$(30): CSR 0,0: DSI
20 CSR 0,0
30 FOR N=1 TO 50
40 LET A$(N)=SPK$
50 NEXT
```

Fig. 9.2. Adding a cursor to DSI.

CHR$(3Ø), and because a PRINT instruction causes a new line to be selected, we have to restore the cursor by using CSR Ø,Ø before the DSI command. It's an unusual and useful way of getting some text into the form of a string, and since the Memotech allows very long strings (provided that you dimension them correctly), you could fill a whole screen with text and store it in this way. As you'll see, however, there are other ways of doing this.

Another instruction that we haven't used so far is EDITOR. This allows you to restrict the screen in such a way that an input must be correctly arranged. Like INPUT, it will not accept anything that follows a comma, but it is much easier to control than INPUT. For example, suppose that you wanted to have a date typed in the form DDMMYY, meaning two digits each for day, month and year, like 121283 (12th December, 1983). This is a perfect example for EDITOR, and Fig. 9.3 shows how the input is controlled and, incidentally, how to unscramble the date from the figures. The first part of the program simply gives instructions, and the next line, line 4Ø, forms a virtual screen. The EDITOR command must make use of VS Ø, so we form a screen with this number. It's a text screen, and we make the starting position at X = 16, Y = 6. The next figure is for the number of characters. Now, we have 6 characters to type, but we need another one for the RET key, so we have to specify 7. One line only is provided for, and since this is a text screen, the figure of 4Ø has to be placed at the end.

When the EDITOR instruction is obeyed, the screen clears to black and white. You could change this by using different PAPER and INK for VS Ø. You can then type in the date. If you make a

```
10 CLS
20 PRINT "Please type date in the form-
"
30 PRINT : PRINT "DDMMYY -such as 03078
4"
40 CRVS 0,0,16,6,7,1,40
50 EDITOR D$: CLS
60 VS 5: CLS : CSR 1,5
70 LET DAY=VAL(D$(1,2))
80 FOR J=1 TO VAL(D$(3,2))
90 READ MON$: NEXT
100 PRINT "Date is ";DAY;" of ";MON$;"
19";D$(5,2)
500 DATA  January, February,March,April
,May,June,July,August,September,Octobe
r,November,December
```

Fig. 9.3. Using the EDITOR instruction.

mistake, just type an extra figure, and you will see the previous entry vanish! When the date looks correct, press RET, and the rest of the program will run. You have reached the stage at which I don't have to spell out how that last line is obtained! The use of EDITOR is a very neat trick when a user is likely to make a lot of mistakes, because you can change your mind as much as you like until you press RET. Having the restricted screen area for entry helps you to concentrate on what you are doing, though it can be a curse if you forget how many letters you are allowed and you find that one more letter wipes out the lot!

NODDY tricks

The Memotech comes with a special programming language for text. This is not a true word-processing program but it does allow you to organise text in a way that would otherwise need a lot of fancy programming in BASIC. The NODDY language is designed to work along with BASIC, so you can use NODDY to create text that will later be used in a BASIC program. This may be used for nothing more than providing instructions for a program, but it can be expanded to form a much greater part of a program such as an 'Adventure' type of game which uses text extensively. The text that is created using NODDY can be saved by itself, if you are using NODDY alone, or along with a BASIC program if you are mixing

the two. You can also print the NODDY text on a printer such as the Epson which I am using at the moment.

We'll start by creating some text. Unlike BASIC, where I can print a listing, and you can then type it for yourself, I have to describe the Noddy actions, because some of them are not 'listed' when text is printed. Ideally, then, you should read this as you sit at the keyboard, and be prepared to experiment on the way. Start by calling the NODDY language, which means typing NODDY and then pressing RET. The result of this is that a prompt, 'Noddy>' appears at the bottom left-hand side of the screen, with a flashing cursor. This is a reminder that NODDY has been called. Since NODDY creates text in screenfuls, or 'pages', you have to start by giving each page a filename. Unlike BASIC, in which you need a filename only when you want to save a program on tape, Noddy needs a filename for each page *before* you create the text. This is what makes it possible to call up a page of text at any time.

A good logical start looks like P1, short for Page 1. You could, of course, type Page 1 if you like. Type P1 or Page 1, and the press RET again. You will now see your title, P1 appear at the top left-hand side of the screen, with the cursor flashing next to it. You can now create text on the screen, and it's up to you! The cursor can be moved around the screen by using the arrowed keys on the right-hand keypad, so I suggest that you move it down and to the left so that the cursor is under the '1' of P1. Now you can start with text. There are two important things to note now. One is that you *must not press the CLS key*, because this will erase everything on the screen. The other is that you must not press the RET key until you are certain that you have finished with a complete page. Your first effort at text creation might not look very good.Figure 9.4 shows what my first effort looked like, with no attempt to make it look better! I'll deal later with how this was printed – the important point at the moment is that this is *exactly* how the text appeared on the screen when I pressed the RET key.

Now when the RET key has been pressed, the 'Noddy>' prompt appears again at the bottom left-hand side. You then have the choice of creating a new page of text, returning to the old one, or creating a control program. To create a new page of text, you type a new and different filename, like P2, and press RET. The screen will clear, and you can then type another page in the same way as you typed the first one, ending when you press the RET key again. If you want to go back to the first page to edit it, then you have to use its filename. Type P1 (or Page 1 if that was what you used), and press RET. This

```
P1

 This is my first attempt to use NODDY
to create text. I can use the arrowed k
eys to move the cursor, and the BS key
to erase text. So far, I have not attem
pted to make the text neat, and words a
re beìng split at the end of lines. Thi
s can be overcome by editig, but I want
 you to ssee the text in its raw state
first. The previous line shows a proble
m with this keyboard, that of keybounce
, because one press of the s key has re
sulted i two letteers. It's just done i
t again! Now I shall press RET too end
the page.
```

Fig. 9.4. A first attempt to create text with NODDY. The faults can be corrected with editing.

will place your page on the screen just as it was when you left it by pressing RET. You can now edit, using all of the editing keys in the usual way. Figure 9.5 shows the edited version of Fig. 9.4, with the words that were split placed on to the next line, and some changes in the text. Normally, you would carry out this correction as you created the text, because editing afterwards is not quite so satisfactory. The reason is that when you insert a word, as you may have to when correcting a split word, you may push another word

```
P1

 This is my first attempt to use NODDY
to create text. I can use the arrowed
keys to move the cursor, and the BS key
to erase the text. So far, I have not
attempted to make the text neat, and
words are being split at the end of
lines. This can be overcome by editing,
but I want you to see the text in its
raw state first. The previous line
showed a problem with this keyboard,
that of keybounce, because one press of
the s key has resulted in two letters.
Now I shall press RET to end the page.
```

Fig. 9.5. The edited text.

off the edge of the screen. A true word-processing program would be arranged so that this word was shifted on to the next line, but NODDY does not do this. There is a danger, then, unless you have a printed copy, of losing your way when you edit afterwards. You may find, for example, that you have lost so many words from a line that you can no longer remember what the line was about. If you think that this is likely, one way around it is to press ESC, then 'I'. This will insert a blank line, so that you have more space to put new text in the middle of old text. Because there is a blank line available (and you can use ESC I to make another one if you want), no old text has to be shifted or deleted to make way for the new. Figure 9.6 lists the editing commands that can be used with NODDY text.

---

(1) Arrowed keys will move cursor in arrow direction. These keys have 'wrapover', so that if you keep your finger on the up-arrow key, for example, you will send the cursor off the top of the screen and it will then appear at the bottom.

(2) BS will back-space, the same action as left-arrow.

(3) EOL key will remove text from the cursor position to the end of that line on the screen.

(4) ENT (SHIFT CLS) has the same effect as RET.

(5) HOME key places cursor at top left-hand corner.

(6) DEL deletes the character under the cursor, and shifts the next character from the right so that it is now under the cursor. You can keep DEL pressed to delete several characters. You can delete the page title (like P1) if you like.

(7) INS pressed once allows you to insert a character at the cursor position, instead of replacing it. The action will continue until you press INS again, or press RET.

(8) TAB moves the cursor eight places to the right.

(9) ESC, then I inserts a whole line.

(10) ESC, then J deletes a line, or a line space.

Fig. 9.6. The NODDY Editing commands.

Control yourself

Suppose that you have created some pages of NODDY text, with

filenames P1, P2 and P3. We'll assume that you have edited and corrected these pages so that they are organised in just the way that you want to see on the screen. The next thing that we have to look at is how we can control this display. This is done by creating a control program, which is another NODDY page. It's this control program which makes NODDY so useful, because it can be used to organise how the text appears, and it can be called up, like a subroutine, from BASIC.

The control program can be written in the same way as any other NODDY page. In other words, we start with the 'Noddy>' prompt showing, and we type a distinctive filename for the control program. We might be shamelessly unoriginal and use PROG. Type this, then press RET, and the name shows once again at the top of the screen. You can then type a set of NODDY instructions. These can be in a line, with spaces between instructions, or in a list, like BASIC lines. NODDY does not use line numbers (BASIC is one of the few languages that does), but it marks the start of each new instruction with an asterisk. The instructions can be full words, or just the first letter, because NODDY uses only a few command words. These, and their abbreviations are listed in Fig. 9.7.

Command	*Effect*
*A	Advance to next page on stack.
*B label	Go to label letter or word.
*D page	Display named page.
*E	Enter input.
*G	Go to page, can specify label.
*I symbol.label	If input matches symbol, go to label.
*L page	Print NODDY page on printer.
*O	Remove page from stack.
*P	Pause for one second.
*R	Return to BASIC.
*S page, page	Stack up pages.

Fig. 9.7. List of NODDY command words.

The three that we need at the moment are *D, *P and *R. *D means *DISPLAY, and it will have the effect of displaying a NODDY page. Which page? You have to specify the page by its filename, followed by a full stop. For example, you might use *D P1. to order a display of the page P1. The full stop is *very important*, because it marks the end of the filename. If NODDY has been called from a BASIC program, you might want to hold the page on the

screen for some time before returning to BASIC. The *P command causes a one-second pause, and you can type as many *P's as you like. Finally, the *R command causes things to return to normal. Normal will mean BASIC, because the control program is the way in which we arrange for BASIC to call up a NODDY program. You might want to do this if these were pages of instructions, and you wanted to call them from a BASIC program, perhaps by typing 'HELP'. Suppose, for example, that you had a NODDY control program which looked like the one in Fig. 9.8. This calls for a display

```
PROG
     *DISPLAY P1.
     *PAUSE *P
     *P *P *P
     *R
```

Fig. 9.8. Example of a NODDY control program.

of page P1, five seconds pause, and then a return to BASIC. The next step is to see how this is called from BASIC. First of all, however, you need to know how to return to BASIC after writing a program in NODDY. You do this by typing DIR (*not* dir), then pressing the CLS key, then pressing RET. DIR is the command that you use to show what NODDY programs are present, in case you have forgotten their filenames. As you might guess, DIR is short for 'DIRectory', and the command is also used in some disk systems for the same purpose.

Figure 9.9 shows how NODDY can be called from BASIC. Line

```
10 CLS
20 INPUT "YOUR COMMAND_ (OR HELP) ";REP
LY$
30 IF REPLY$="HELP" THEN  PLOD "PROG" E
LSE  GOTO 50
40 PRINT "Does that help you?": GOTO 60
50 PRINT "Your command is ";REPLY$
60 STOP
```

Fig. 9.9. Calling NODDY from BASIC.

2Ø asks you for a command, such as you might need in an 'Adventure' game. You can get instructions by typing HELP (not help!). Line 3Ø then makes the selection, and the command which runs NODDY is 'PLOD'. This has to be followed by the filename of the *control program*, rather than the page P1. The effect is to carry out the display, pause, and then carry out line 4Ø. Note that the text

in line 4Ø will join to the end of the NODDY text unless you add an extra PRINT, or shift the cursor, or clear the screen.

Much more control

Many programmers will want no more of NODDY than some text editing, and instruction lines for BASIC programs. What we have done up to now is therefore adequate for these needs. Remember that the NODDY programs are saved with the BASIC, but are not listed with it. To see your NODDY programs, you have to enter NODDY, and then type the filename of the page that you want to see. If you have forgotten the filenames, then type DIR (then RET) to display them. The important point is that you use CLS to return to BASIC *only* after using DIR. Its use at any other time will completely remove the page that is displayed.

The way that NODDY can use a control program to decide what is shown on the screen can, however, be greatly extended. As well as placing NODDY pages on the screen, the control program can accept inputs from the keyboard, make decision steps, and perform loops. These actions are illustrated in the manual, but some more extended explanations and examples may prove useful to you, particularly if you want to make use of NODDY in writing question and answer games or educational programs.

Start, then, by adding more pages. I shall assume that you have P1, P2 and P3 available, so type P2 and P3 if you have not already done so. Keeping to the filename of PROG for the control program, extend it to the scheme which is shown in Fig. 9.10. This displays the

```
PROG
     *DISPLAY P1.
     *PAUSE *P
     *P *P *P
     *D P2. *P *P *P *P
     *D P3. *P *P *P *P
     *R
```

Fig. 9.10. An extended NODDY display of three pages.

first page, pauses for five seconds, then displays the second page. This is displayed for four seconds, then replaced by P3. As usual, there isn't much reading on P3, so it lasts just four seconds, and then returns you to BASIC.

Now this is a simple sequence, but if you had to program lines in

BASIC to carry this out, it would have taken up quite a lot of program lines, and a considerable amount of effort. Even a simple sequence like this illustrates how easy it is to use NODDY to display pages. I keep typing the word 'pages', but none of them need be a full page. Some of these 'pages', in fact, may consist of nothing more than a headline – it's entirely up to you. The advantage of using NODDY is that you can see what the page will look like as you type it. When you create text by using BASIC PRINT lines, you can't actually see what the text looks like on the screen until you RUN it, and then you may have a lot of tedious editing to do.

In that example, one page followed another after a preset time interval. This is not always desirable, and we often want to give the user the chance to decide when to change pages. NODDY provides for this by means of an ENTER(*E) command. The *E command operates like INPUT in BASIC, so that nothing happens until the RET key is pressed. Figure 9.11 shows the PROG control program altered so as to allow you as long as you like to inspect pages. You

```
PROG
     *DISPLAY P1.
     *E
     *D P2.
     *E
     *D P3.
     *E
     *R
```

Fig. 9.11. Using *E to give you as much time as you need to read each page.

can modify your existing version of PROG by using ESC J to delete a line and ESC I to create a blank line. Remember that ESC J means pressing the ESC key, releasing it, and then pressing the J key.

What if you wanted to choose one of the pages, rather than seeing all of them in sequence? NODDY provides for this by allowing a decision step rather like the use of IF in BASIC. Now, if you have programmed in languages other than BASIC, some of the ideas that are used here will be familiar. If you are a newcomer, or have only programmed earlier models of computers that used BASIC, then all of this will be new to you. To start with, since we have no line numbers in NODDY, we have to refer to a line or instruction that we want by using a *label*. A label is a letter, and so that the machine can tell the difference between a letter that is being used as a label and one which is part of filename, a label *must* start with the ^ mark. Valid labels would be, for example, ^a , ^z , ^B , ^Loop , the

important point is to start the label name with the ^ sign. A label like this is placed just before the instruction to which it refers. Instead of an instruction like GOTO 2ØØ, then NODDY can use a reference to the label instead.

Figure 9.12 illustrates a selection program written in NODDY, and using these principles. The first line starts with the label ^z, and

```
PROG
     ^z *E
     *IF P1,a
     *IF P2,b
     *IF P3,c
     *IF R,r
     *B z


     ^a *D P1. *B z
     ^b *D P2. *B z
     ^c *D P3. *B z
     ^r *R
```

Fig. 9.12. Menus and loops in NODDY.

contains the *E command. This line will therefore cause the program to wait for an input, and will do nothing until the RET key has been pressed. The next four lines are selection lines. I have, incidentally, shown these in neat columns, but you could type them in one continuous line, with spaces between commands, if you wanted to. The key command word is *IF, and it has to be followed by two items. The first of these is the letter or word that you expect the user to type. In the first of these lines, we are looking for the user to type 'P1', and then press RET. If this is done, then the 'a' that follows P1 in the instruction will cause the program to look for a label 'a', and execute everything that follows 'a'. The next line, *IF P2,b means that if P2 has been typed and RET pressed, then the program will jump to label b. If one test fails, then the program will move to the next one, so if you type P3, then the program tries the P1 line, then the P2 line, and then the P3 line which will send it to label 'c'. The last choice, *F R,r, allows you to return to BASIC by typing R. At the end of this piece of the program, we need the *BRANCH instruction, *B z. This makes the program go back to the label 'z', which is the start at *E. This ensures that when a choice is incorrect, like typing C or p1 (rather than P1), then the program returns to the *ENTER step. Without this stage, the first labelled line would be

run by entering *any* letter, which is not exactly what we intend.

Under this set of lines, we have arranged the choices. Label 'a' is followed by the commands *D P1 (display P1) and *B z . This will cause NODDY to display page 1, and the branch back to the start at label 'z'. If you omitted the *B z section, P1 would be displayed momentarily, and then replaced by P2, because, as in BASIC, instructions are carried out in sequence. The other choices of displayed pages are also followed by a branch back to the start, but the final 'r' label simply contains the *R return which causes a return to BASIC.

These NODDY commands allow you to set up quite useful programs which will run within a BASIC program, and save a lot of BASIC programming. The Memotech manual goes quite a lot further into the other commands of NODDY, but it's unlikely that you will need these until your programming has reached a much more advanced level. When you need to use these commands, the explanations that are in the manual, along with your experience, will be enough for you to make sense of NODDY for more advanced work. For the moment, then, this is as far as we go, and the next stop is a whistle-stop, a look at the sound commands.

Chapter Ten

Sounding Out The Memotech

The ability to produce sound is an essential feature of all modern computers. The sound of the Memotech comes from the loudspeaker of the TV receiver that you use to see the display, so you have more control over the volume of this sound than is possible with a lot of other computers. In addition, a socket on the back of the Memotech allows you to connect to a hi-fi amplifier unit, so that you can hear the sound at greater volume, and with better bass (low notes). The difference is quite astounding if you have only ever heard the sound delivered from the tiny loudspeaker of a TV receiver. If you use a colour monitor for high-quality pictures, you will have to obtain your sound signals from the hi-fi socket and use a separate amplifier. This is because colour monitors generally do not have loudspeakers or amplifier circuits.

Before you start to work with the sound commands of the Memotech, however, you must ensure that your TV is correctly tuned for sound. Turn up the volume control of the receiver and listen. You will probably hear a rasping sound, a low buzzing note. Tune the TV until this sound is at its minimum. Unfortunately, there are TV receivers which do not give the best possible picture when the sound is correctly tuned, and which do not give the best sound when the picture is correctly tuned. The problem is that when you replace the cover over the tuning controls of a modern TV, the tuning is then automatically controlled, and it will always tune to the best picture, which may not be the spot for the best sound. What it all boils down to is that TV receivers are not ideal for use with computers – but you know that already!

What we call sound is the result of rapid changes of the pressure of the air round our ears. We don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or hertz. A cycle of a wave is a set of changes of pressure, first in one direction, then in the other and back to normal, which we

can illustrate by the graph in Fig. 10.1. The reason that we talk about a sound *wave* is because the shape of this graph is a wave shape.

The frequency of sound is its number of hertz – the number of cycles of changing air pressure per second. If this amount is less than

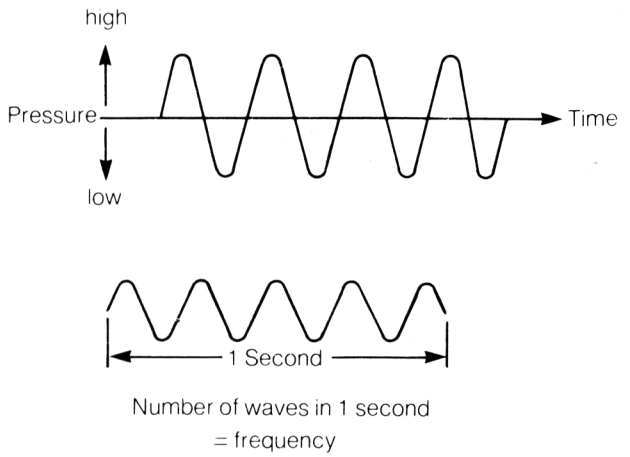


Fig. 10.1. Sound waveforms, showing how the air pressure changes with time. The number of changes per second is called the frequency.

about 20 hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 hertz, going up to about 15000 hertz. The frequency of the waves corresponds to what we sense as the *pitch* of a note. A low frequency of 80 to 120 hertz corresponds to a low-pitch bass note. A frequency of 400 or above corresponds to a high pitch treble note.

The amount of pressure change determines what we call the loudness of a note. This is measured in terms of *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

For the purposes of writing music, a composer has to specify how loud each note will be, for how long it has to be played, and its pitch. In written music, loudness is indicated by letters such as *f* (loud) and *p* (soft), and by using more than one letter if necessary. For example, *fff* means very loud, and *ppp* means very soft. The duration of a note is indicated in two ways. One of these ways is a metronome reading. A metronome is a sound generator which ticks at precise intervals,

and a metronome reading indicates how many *beats* (units of notes) are sounded per minute. The unit duration of note is called the *crochet*, so the metronome reading decides on the time of a crochet by measuring the number of crochets that would be sounded per minute. The durations of the other notes are then measured in comparison to this. A *minim* sounds for twice as long as a crochet, and a *semibreve* sounds for twice as long as a minim, which is four times as long as a crochet. The *quaver* is allowed half the time of a crochet, and a *semiquaver* a quarter the time of the crochet. These differently timed notes are indicated by the shape of the symbols that are used for the notes (Fig.10.2). In addition, there are symbols for

Symbol	Time	Name
	1/8	Demisemiquaver
	1/4	Semiquaver
	1/2	Quaver
	1	Crotchet
	2	Minim
	4	Semibreve

Adding a dot after a note lengthens it by 50%

Fig. 10.2. The symbols that are used in written music to indicate the time of each note.

different durations of silence in the music, and these are illustrated in Fig. 10.3. Some music scores do not show a metronome reading, but rely on the use of Italian words to indicate the time of a crochet less precisely.

Rest Symbol	Time
	1/4
	1/2
	1
	2
	4

Fig. 10.3. The symbols for silences in written music.

The pitch of a note is indicated in written music by placing it on a kind of musical 'map' that is called the *stave*. Piano music shows two of these staves, each consisting of five lines and four spaces. The set of lines which is marked with the sign like the ampersand (&) is the *treble stave*, used for the higher notes, and the stave below it is the *bass stave*. The bass stave is also marked with a distinctive symbol, like a reversed 'C'. When music is written for instruments which do not use a keyboard, then only one stave is used. Piano and organ music always shows two staves, with a place for a note placed between them. This note is called *Middle C* and on a piano it is played by pressing the key which is almost exactly at the centre of the keyboard. Figure 10.4 shows the staves with the notes marked.

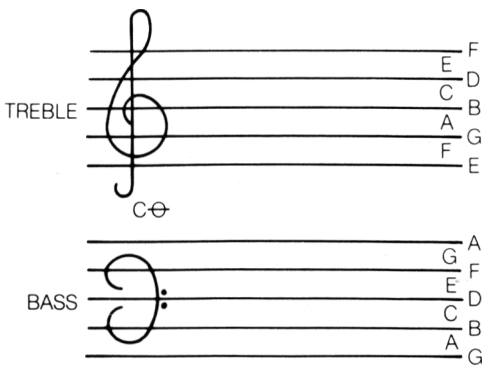


Fig. 10.4. The staves, with the names of the notes written in.

The notes that are shown in Fig. 10.4 are arranged in groups of eight, counting inclusively. This group is called an *octave*. Music from the Western hemisphere traditionally uses a total of twelve distinct notes in an octave, and this full range is illustrated in Fig. 10.5, which shows the appearance of part of the piano keyboard. The half-pitch notes, or *semitones* are marked on the piano mainly by the black keys, though two semitones (between B and C, and between E and F) are not marked in this way. On written music, semitones are indicated by using the signs # (sharp) and ♭ (flat). A

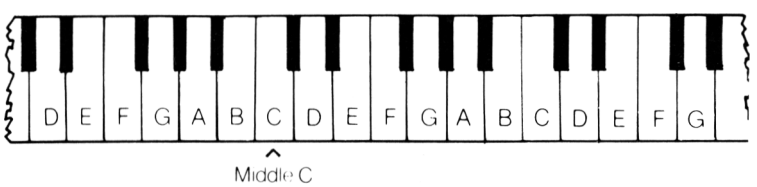


Fig. 10.5. Part of the piano keyboard, showing Middle C. There is only one semitone between B and C, and between E and F.

sharp indicates that the pitch has to be raised one semitone above the written note, and a flat indicates that the pitch is to be lowered one semitone below the written note. On the piano, the semitone above one note is the same as the semitone below the next note, so that C# is the same as D♭. This is not necessarily true on other instruments.

The Memotech SOUND

To work, then. The Memotech provides a sound command which can be used either as a simple command, or as a complicated one. How you use it depends on what you want of it, so that you don't have to do a lot of planning just to produce a warning note. When you are experienced in the use of the sound command, however, you can make use of it in much more interesting ways. We shall start with the straightforward production of notes.

This is provided by the SOUND instruction. The SOUND instruction has to be followed by three numbers. Of these, the first number is a *channel number*. The Memotech can produce four notes of sound at the same time, and all four notes can be separately controlled. This is done by allocating each note to a separate *channel*. These channel numbers can range from ∅ to 3; the numbers then repeat, so that using 4 has the same effect as using ∅, using 5 has the same effect as using 1, and so on.

The second number that follows the SOUND instruction is the *pitch number*. This controls the pitch of the note that is produced by the Memotech, and its range is from 1 (highest pitch note) to 1023 (lowest note). You can actually use numbers up to 65535, but you get the same scale of notes for each set of 1024 numbers. The musical equivalents are given in Fig. 10.6, which shows the pitch numbers that correspond most closely to written notes and also to the piano scale. The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of notes, called the 'scale of C Major'. The scale starts on a note that is called Middle C, and ends on a note that is also called C, but which is the eighth note above Middle C. A group of eight notes like this is called an *octave*, so that the note you end with in this scale is the C which is one octave above Middle C.

The third number in the SOUND instruction is one that sets the relative volume of the note. Relative volume means that if you use different values of this number with the same setting of the volume control of the TV receiver, you will hear different volumes of sound.

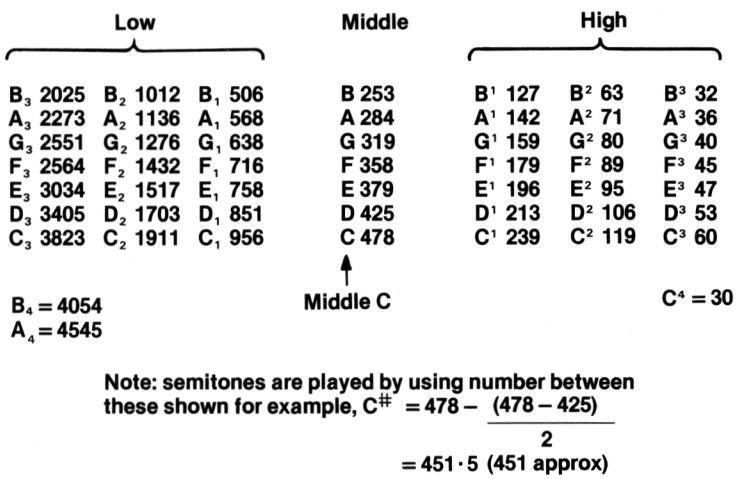


Fig. 10.6. Pitch numbers for the Memotech. This list covers all the white notes of the piano.

You can set the *absolute* volume as you wish by using the volume control of the receiver as usual. The range of values for this number is ∅ (silence) to 15 (full volume). Numbers greater than 15 will not cause an error message, they repeat like the channel numbers, so that 16 causes the same volume (zero!) as ∅, 17 is the same is 1, and so on.

Let's start our investigation of the SOUND instruction with a simple single note. This is illustrated in Fig. 10.7, which has the main SOUND instruction in line 1∅. Channel ∅ is used, and the pitch of

```
10 SOUND 0,480,15
20 PAUSE 1000
30 SOUND 0,0,0
```

Fig. 10.7. Playing a single note. This will continue until it is stopped.

the note is selected by the number 48∅, which gives a note very close to Middle C. A volume of 15 is used. You could have used channel numbers 1 or 2 for this purpose, but channel 3 is different, as we'll see later. Line 1∅ turns the sound on, and the action of this simple SOUND command is that the sound will continue until something happens to turn it off. To control how long the sound lasts, we use a delay in line 2∅. The turn-off instruction is in line 3∅, in the form of SOUND ∅,∅,∅. This is the method of turning off the sound that we shall use throughout this chapter. Pressing the SYSTEM RESET also turns off the sound, but that's a desperate measure, because it also removes your program!

The next step is to try a musical scale. The table of numbers and

frequencies that is shown in the Appendix of the Memotech manual (provisional copy) does not show how the frequencies are related to musical notes, so you will have to rely on the scale of Fig. 10.6. Making use of this figure, we arrive at the program in Fig. 10.8, which plays the scale of C Major. This is a scale which starts with

```
10 FOR N=1 TO 8
20 READ F
30 SOUND 0,F,15
40 PAUSE 1000
50 NEXT
60 SOUND 0,0,0
100 DATA   478,426,379,358,319,284,253,
239
```

Fig. 10.8. The scale of C Major, by Memotech!

Middle C, and goes up to the C above it. In this scale of notes, the frequencies are such that the C above Middle C has exactly double the frequency of Middle C itself. The same is true of all the other notes in the scale – doubling the frequency is equivalent to going up one octave in pitch. Halving the frequency corresponds to going down one octave in pitch. The Memotech pitch numbers operate the opposite way round, so that an octave rise in pitch is achieved by halving the Memotech number. You can see from Fig. 10.8 that Middle C uses a number of 478, and the C above it uses 239, which is exactly half. We can't always achieve this when we use only whole numbers, but we keep as close as possible to these ratios.

The next step is to investigate the use of more than one channel of sound at a time. Figure 10.9 shows a chord being sounded with three channels. A volume number of 15 has been used on each channel for

```
10 SOUND 0,478,15
20 SOUND 1,379,15
30 SOUND 2,319,15
40 PAUSE 1000
50 FOR N=0 TO 2: SOUND N,0,0: NEXT
```

Fig. 10.9. A chord which uses three channels.

simplicity, but for the best musical effects you might want to use different volumes for different channels. Because the human ear is less sensitive to low notes and high notes as compared to notes around Middle C, it's often useful to sound these notes at higher volume numbers than the notes in the middle range. The sound is

turned off in this example by using a loop in line 5ø which sets all the numbers to zero in all four channels.

Music, music, music

The ability of the Memotech to produce more than one channel of sound allows you to compose music which has harmony. Unless you are an accomplished composer, or want to be, it's best to use sheet music as a guide. The best music to use, if you want three channels, is violin and piano, or soprano voice and piano, or flute and piano. These provide a range of notes that most TV loudspeakers can cope with reasonably well. Music for the double-bass does not sound good on a small speaker!

The best technique to use is a loop for the number of notes and silences that you will use. The PAUSE which causes a silence should be programmed with a variable rather than a number, so that you can speed up or slow down the music without having to change every line. When you try this at first, it's advisable to have one line of DATA for each note. If you want to keep a note playing in one channel while other notes change, you will have to break out of the loop and use separate SOUND instructions. You'll find that piano music usually sounds better if you have short pauses between notes,

```
10 FOR N=1 TO 9
20 READ C1,C2,C3,P
30 SOUND 0,C1,15
40 SOUND 1,C2,9
50 SOUND 2,C3,8
60 PAUSE P*100
70 FOR J=0 TO 2
80 SOUND J,0,0: PAUSE 25
90 NEXT : NEXT
500 DATA  478,319,956,7
510 DATA  506,319,851,7
520 DATA  478,319,956,21
530 DATA  426,319,851,7
540 DATA  379,253,758,14
550 DATA  319,268,638,7
560 DATA  358,284,716,7
570 DATA  379,319,758,10
580 DATA  426,319,851,14
```

Fig. 10.10. A touch of harmony.

but that organ music doesn't need this. Experience is the main thing that you need once you have acquired the programming skills.

As an example, look at Fig. 10.10, which illustrates a bit of three-part harmony. This was not written with the aid of sheet music, and I had to operate by writing one channel at a time. This is comparatively straightforward, because the main program consists of a loop, with C1, C2, and C3 used for the channel notes in channels Ø, 1 and 2 respectively, and P used for the pause. Using this form makes it easy to change the tune by altering the DATA lines. I wrote the numbers for channel Ø first, with zeros for the other channels. This was adjusted as necessary, and the multiplier for the PAUSE set to a value that gave reasonable timing. Having got the tune as I wanted it, I added the channel 1 sound, and when that was sorted out, I added the channel 2 sound. It's not as good as it would be if I had used a score. Beethoven wrote the score, I only programmed the computer.

Special effects department

The SOUND instruction, even in this simple form, can produce a large range of useful sound effects. Let's start with a rising pitch of note which makes a useful warning, or a 'something about to happen' note. This is illustrated in Fig. 10.11. The loop that starts in

```
10 FOR N=1023 TO 1 STEP -1
20 SOUND 0,N,15: NEXT
30 SOUND 0,0,0
```

Fig. 10.11. A warning note, of increasing pitch.

line 1Ø uses values of J that range from 1Ø23 to 1, the full range that the SOUND instruction uses. These are the numbers that we shall use as pitch numbers in the SOUND instruction in line 2Ø. Line 3Ø switches off the sound at the end of the loop. If you want to use more than one of these rising notes, then all you have to do is to use a starting number that is greater than 1Ø23. Try 65535, for a long warning. This is easier than programming another loop! The reason for the action is that the command is organised to use numbers up to 1Ø23 only, and when you use a larger number, what operates the command is the *remainder* after the number has been divided by 1Ø24. For example, if you use 2ØØØØ, then because 2ØØØØ / 1Ø24 = 19 and 544 remainder, then this is equivalent to using 544 in the SOUND command.

Figure 10.12 shows a program that produces a warbling note. This is particularly useful for attracting attention, or for announcing an event in a game. For some reason, a warbling note attracts our attention more than a single note, which is why a warbling note was

```
10 FOR J=1 TO 200
20 SOUND 0,239,15
30 PAUSE 20
40 SOUND 0,244,15
50 PAUSE 20
60 NEXT
70 SOUND 0,0,0
```

Fig. 10.12. A warbling-note program.

chosen for the later types of telephones. The warble in this program uses the loop that starts in line 1Ø. This sounds 200 pairs of notes, which are short with a duration set by the short time delays in lines 3Ø and 5Ø. The two pitch numbers that have been chosen in this example are 239 and 244. Higher pitches are even more effective, and values like 9Ø and 95 give effective attention-getting warbles.

The noise channel

Channel 3 of the Memotech is different, because what you get from it is not ordinary musical notes, but noise. What we call noise is a random mixture of frequencies, but very often you find that one frequency or frequency range is much louder than others. A noise channel is useful because it can be used to provide effects like surf, drumbeats, and other sounds that are otherwise impossible to produce with the ordinary SOUND command.

Figure 10.13 illustrates channel 3 in action. The number that we

```
10 FOR N=0 TO 7
20 CSR 5,6: PRINT N
30 SOUND 3,N,15
40 PAUSE 2000: NEXT
50 SOUND 3,0,0
```

Fig. 10.13. Using the noise channel. Only a small range of 'pitch' numbers can be used.

use for 'frequency' does not have the same effect as it has in the other channels, nor the same range. The range is Ø to 7, and the effects simply repeat if you use numbers greater than 7. You'll find that

numbers 4, 5 and 6 are the most useful for noise effects. If you want an interesting bit of accompaniment, though, try a range of Ø to 1ØØ in the loop, and make the pause number 5ØØ! For a better appreciation of the noise channel, however, try the program in Fig.10.14. This gives a splendid 'boing-boing' effect that is caused by starting each note at maximum volume and then decreasing the volume in a loop. Try this one with the number 5 and then number 6

```
10 FOR J=1 TO 10: FOR N=15 TO 0 STEP -1
20 SOUND 3,4,N
30 PAUSE 50: NEXT : NEXT
40 SOUND 3,0,0
```

Fig. 10.14. The 'boing-boing' effect.

in place of 4 in line 2Ø to hear the effect. Unlike text and graphics programs, the effects of a sound instruction are almost impossible to describe, and you simply have to type and listen. It's a good idea to make an index of sound effects, using your cassette recorder to tape the sounds as well as the programs. Remember that the Memotech sends sound signals from its 'hi-fi' socket, and these signals can be sent to a cassette recorder. You'll need a suitable connecting cable, with the right type of plugs, though.

Waveshapes unlimited

It's time now to investigate the effect of extending the SOUND command so that it controls the *envelope* of the sound that you hear. The effect is anything but simple, and because it's difficult to describe in words what a noise sounds like, you simply have to try the programs and listen! First of all, I have to explain what is meant by an 'envelope'. The sounds that the simple SOUND command produces have a constant amplitude and constant frequency. In simpler words, their loudness is constant and so is their pitch as the notes play. Musical instruments, however, produce notes in which the loudness varies in each note. A piano note, for example, is loud at the instant it is struck, because that's when the hammer strikes the strings. This dies away rapidly, and it's the way in which the note's loudness dies away that makes a piano note so distinctive. Other instruments produce notes which behave quite differently, and all instruments produce notes which consist of a mixture of frequencies. That's why you can tell a piano from a violin from a

flute from an oboe, even if they are all playing the same note. A graph of the amplitude of a note, plotted over the time that the note takes, is called the 'envelope' of the note.

Take a look at Fig. 10.15. This shows an envelope which is typical

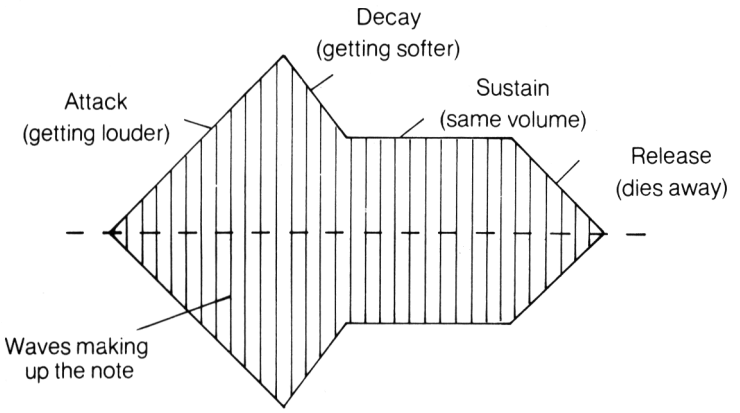


Fig. 10.15. The 'envelope' of a sound.

of many musical notes. It has a sharply rising amplitude at the start, which we call the *attack*. This is followed by the *decay* portion, in which the amplitude drops. The amplitude then remains steady for a time, in the *sustain* section, and then finally drops to zero in the *release* section. The Memotech gives us the chance to synthesise even a waveform like this by using an extended version of the SOUND command. We need, however, one SOUND command for each section of the envelope, so it's easier to make use of simpler shapes if we can.

The simple version of the SOUND command uses just three numbers, but the full version make use of four more. The first of these is a *frequency increment* number. If you specify a number in this position, the frequency of the note will be changed by this amount each one-sixty-fourth of a second. If you use ∅ here, the frequency will remain constant. The next additional number is the *amplitude increment*. The number that you use here is added to the amplitude of the sound each 1/64 second. You can use a positive number to get increasing amplitude, and a negative number to get decreasing amplitude. The next additional number is a duration number. This refers to the duration of the *change* only, not to the duration of the note. Once again, this is in 64ths of a second. The final number is a *mode* number, which can be ∅ or 1. When this is 1, the sound will start with the amplitude and frequency that are

specified by the second and third numbers of the ordinary SOUND command. If you make the mode number equal to zero, however, these amplitude and frequency numbers will be ignored, and the command will work on whatever values happen to be present. This is useful for synthesising envelopes, because though we may know what the amplitude and frequency are to be at the start of a note, we may not know what values they will have at the point where the envelope changes.

Sealing the envelope

Though the rules for using the envelope numbers appear quite straightforward, there are a lot of hints, tips and conditions which are not obvious from the manual. To start with, the use of the extended SOUND command means that you can use a completely different range of numbers for pitch and volume control. The new numbers for pitch are eight times as much as the old ones, so that the range of useful pitch numbers is from Ø to 8192. The range of volume numbers is stated as from Ø to 24Ø, sixteen times as much. This has been done so as to allow smaller changes in pitch or volume. When you change the pitch or volume sixty-four times per second, each change has to be small, otherwise the changes are too rapid for the ear to follow.

The next point follows on from that one. Any note in which you want to make changes should be a fairly long-duration note, unless you are trying to create noises like pistol-shots. Certainly for musical notes, short duration envelopes will prove to be very disappointing. As a general rule, try to aim at musical notes which will last for more than a second. The fact that the duration can be controlled by the extended SOUND command should not lead you to expect too much. The duration number controls only the time for which the note *changes*. You still have to issue a command to stop the note, unless the change has been to zero amplitude!

Finally, there is the use of memory. When you use the extended sound command, the computer executes the lines of BASIC at its usual fast pace. If you have specified, however, that a note is to be held for a couple of seconds, and that there will then be another change for three seconds, the values have to be held in a queue somewhere, waiting. This is because the part of the computer that controls the sound is like a miniature computer in itself, and can take values from the memory and work on them at the same time as the

main system gets on with something else. All that the main system has to do is to interrupt its action sixty-four times a second to issue new instructions to the sound system. Getting back to the queue of instructions, though, these are held in a part of the memory that is called the *sound buffer*. Some memory is automatically allocated to this sound buffer use when the computer is switched on, and this allocation is enough for one extended SOUND command. To be sure that you will have enough storage space for each command, it's wise to allocate two units of sound buffer for each extended SOUND command. If you have five extended SOUND commands, then, you have to reserve ten units of sound buffer. This is done by using the SBUF 1Ø command early in the program.

That's the theory, anyway, so now before your head starts to buzz with instructions, let's hear what the effect of these extended sound commands is in practice. Figure 10.16 is an extended SOUND command in which the amplitude is varied from a starting value of 1.

```
10 SOUND 0,3000,1,0,1,1000,1
20 EDIT 10
```

Fig. 10.16. Illustrating the extended SOUND instruction. In this example, the amplitude is being incremented.

The EDIT 1Ø line has been placed there so that you can make changes and find out their effect quickly. Now when this runs, you will hear the sound increase in sixteen steps around once per second. When the changes stop, you will have to type SOUND Ø,Ø,Ø to stop the sound. This suggests that the volume increment number actually affects the range of Ø to 15 rather than the settings of Ø to 24Ø that are stated as the range in the volume number. Suppose, for example, that you try Fig. 10.17. A setting of 5ØØ as the starting volume causes

```
10 SOUND 0,3000,500,0,1,1000,1
20 EDIT 10
```

Fig. 10.17. Checking the practical ranges of values.

the volume to start at a midway value. This suggests that the actual range of starting value can be Ø to 999, rather than Ø to 24Ø. Only steps of 64 in this number seem to have any effect on the volume. Once more, you have to type SOUND Ø,Ø,Ø to end the sound, or run again.

You can now experiment with this, taking the program of Fig.10.16 as a starter. If, for example, you use 2 as the amplitude increment in place of 1, you will hear the amplitude increase to

maximum *twice*. This is because when the volume control number has reached its maximum value, all of its numbers repeat from the beginning. If you use a figure of 1Ø, the sound starts to get quite interesting, like a music tape played backwards. Most musical notes, you see, do the reverse of this, they start loud and fade. Try Fig. 10.18 for this effect. Now use −5Ø as the volume change number, and

```
10 SOUND 0,3000,999,0,-10,1024,1
20 EDIT 10
```

Fig. 10.18. An envelope which is useful for more musical sounds.

1Ø23 as the time. This will give a sharper set of notes, ending with silence. No need for SOUND Ø,Ø,Ø this time! Now try −15Ø as the amplitude change number. You'll find some really curious effects if you increase the number to larger negative values. Try −5ØØ, for example, then the ultimate value of −32767, which gives the same effects as +1!

Now let's try frequency changes. Figure 10.19 shows what happens with a frequency change number of −1. The changes are

```
10 SOUND 0,3800,999,-1,0,1000,1
```

Fig. 10.19. Using frequency changes in an envelope.

quite slow, so try −1Ø. You'll find that this takes you through the whole range of notes more than once. A value of −1ØØ starts to sound more useful, and −1ØØØ gets you into the realm of weird noises. Now try Fig. 10.20, which combines frequency changes with amplitude changes. These can be even more interesting if you use

```
10 SOUND 0,8000,999,-200,-20,39,1
```

Fig. 10.20. Combining frequency and amplitude changes.

Channel 3 to obtain noise. The numbers for Channel 3 in this type of instruction are 32, 4Ø and 48. Figure 10.21 produces a useful 'boing' sound with the number 32 as 'pitch' number, and by substituting 4Ø, and then 48, you can hear the 'boing' become deeper. Note, by the

```
10 SOUND 3,32,940,0,-20,47,1
```

Fig. 10.21. Using Channel 3 with the extended SOUND instruction.

way that a single 'boing' is produced by having a value of 47 for the duration. This is because we have started with 940 as the volume number, and have 2Ø as the change number (negative). Since 2Ø × 47 = 94Ø, the duration is enough to allow one cycle of change only.

By this time, you will have gathered that there is a limit to the amount of planning that you can do for a SOUND command of this type. Experience is your best guide, together with a good ear for changes of sound. The main topic that we still need to look at is the SBUF command. This is needed if you want to change conditions more than once with more than one extended SOUND command. You will need this, for example, if a SOUND command leaves a note playing. When an extended SOUND command has left a sound playing, you can't switch it off by having a SOUND Ø,Ø,Ø (for example) *in the program*. This is because the SOUND Ø,Ø,Ø will not be stored unless you have provided for this, and when it is executed almost as soon as the extended SOUND has started, it has no effect. If we are going to use more than one extended SOUND command, we must provide a SBUF number. A good start is to allow a figure of 2 for each SOUND command.

Figure 10.22 illustrates this point, along with the use of the mode number. Line 1Ø reserves ten buffer spaces, more than we need. Line

```
10 SBUF 10
20 SOUND 0,3000,0,0,10,100,1
30 SOUND 0,3000,0,0,-10,100,0
```

Fig. 10.22. Synthesising an envelope by using a rising amplitude followed by a falling amplitude. Try this with different rates and times for the two parts.

2Ø sets up a note of increasing amplitude, and line 3Ø decreases it again. In line 3Ø, a final figure of Ø means that the command must take the values of amplitude and frequency that have been left it by the previous command. Now try changing the 'change' values to 5Ø and −5Ø, and the duration value to 2Ø. This gives a shorter note that waxes and wanes in a very satisfactory way. Note that the product of duration number × amplitude increment number is kept at a figure of 1000. This rule of thumb seems to work quite well.

Chapter Eleven
Do It Yourself!

You can get a lot of enjoyment from your Memotech when you use it to enter programs from cassettes that you have bought. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that.

Now, I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of GWR steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings, sound, or other programs that require shapes to move around the screen. What we are going to look at in this section is the database type of program, because it's designed in a way that can be used for all other types of programs.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works than from an elaborate program that never seems to do what it should. The second point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly

when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put it on paper

We start, then, with a pad of paper. For myself, I use a student's pad of A4 which is punched so that I can put sheets into a file. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Yes, I said sheets! Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a fairly simple program, but it will illustrate all the skills that you need. After this, it's all up to you!

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to, it's odds on the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it!

As an example, take a look at Fig. 11.1. This shows a program outline plan for a simple game. The aim of the game is to become familiar with the capital cities of countries around the world. The program plan shows what I expect of this game. It must present the name of a country on the screen, and then ask what the name of the capital city is. A little bit more thought produces some additional points. The name of the capital city will have to be correctly spelled and in upper-case letters. A little bit of trickery will be needed to prevent the user (son, daughter, brother, sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we

Aims:
1. Present the name of a country on the screen.
2. Ask what its capital city is called.
3. The reply must be correctly spelled.
4. User must not be able to read the answer from a listing.
5. Give one point for each correct answer.
6. Allow two chances at each question.
7. Keep a track of the number of attempts.
8. Present the score as a number of successes out of number of attempts.
9. Pick country names at random.

Fig. 11.1. A program outline plan. This is your starter!

should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program the way an artist paints a picture. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show instructions. This looks like the sort of job for which NODDY was designed. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 11.2 shows what this might look like at this stage.

1. Display title, then instructions.
2. Display name of country on the screen.
3. Ask for the name of the capital.
4. Use INPUT for reply.
5. Compare reply with correct answer.
6. If correct, add 1 to score and ask if another one is wanted.
7. If incorrect, allow another try.
8. If second attempt is also incorrect, select another question.
9. Ends when user types N in response to 'Do you want another one?'

Fig. 11.2. The next stage in expanding the outline.

The BASIC foundations

Now at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 11.3 shows what you should aim for at this stage. There are only fourteen lines of program here, and that's as much as you want.

```
10 CLS : PLOD "P1"
11 REM TITLE
20 PLOD "P2"
21 REM INSTRUCTIONS
30 GOSUB 1400
31 REM DIMENSIONS
40 GOSUB 2000
41 REM PLAY
50 GOSUB 3000
51 REM SCORE
60 GOSUB 4000
61 REM ANOTHER?
70 IF K$="Y" OR K$="y" THEN  GOTO 40
80 STOP
```

Fig. 11.3. A 'core' or 'foundation' program for the example.

This is a foundation, remember, not the Albert Hall. It's also a program that is being developed, so we've hung some 'danger – men at work' signs around. These take the form of the lines that start with REM. REM means REMinder, and any line of a program that starts with REM will be ignored by the computer. This means that you can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you LIST. In Fig. 11.3, I have put the REM notes on lines which are numbered just 1 more than the main lines. This way, I can remove all the REM lines later. How much later? When the program is complete, tested, and working perfectly. REMs are useful, but they make a program take up more space in memory, and run slightly slower. I always like to keep one copy of a program with the REMs in place, and another 'working' copy which has no REMs. That way I have a fast and efficient program for

everyday use, and a fully-detailed version that I can use if I want to make changes.

Let's get back to the program itself. As you can see, it consists of a set of GOSUB instructions, with references to lines that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Fig. 11.2 exactly, and the only part that is not committed to a GOSUB is the IF in line 7Ø. What we shall do is to write a subroutine which will use INKEY$ to look for a 'Y' or 'N' being pressed, and line 7Ø deals with the answer. What's the question? Why, it's the 'Do you want another game' step that we planned for earlier.

Take a good long look at this piece of program, because it's important. The use of all the subroutines means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
9 GOTO 3Ø
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it. You will have to remember, however, to write the NODDY pages before you remove this line.

The next step is to get to the keyboard (at last, at last!) and enter this core program. If you use the GOTO step to skip round the title and instructions temporarily, you can then put in simple PRINT lines at each subroutine line number. We did this, you remember, in the program of Fig. 4.15, so you know how to go about it. This allows you to test your core program and be sure that it will work before you go any further.

The next step is to record this core program. You can then build up the rest of the program by adding to the core. If you have the core recorded, then you can load this into your Memotech, type in one of the subroutines, and then test. When you are satisfied that it works, you can record the whole lot on another cassette. Next time you want to add a subroutine, you start with this version, and so on. This

way, you keep tapes of a steadily-growing program, with each stage tested and known to work. If there is a thunderstorm during these operations, causing a power failure, then all you can lose is the new material that has not yet been recorded. You think it can't happen to you? It has certainly happened to me – and only five minutes ago!

Subroutine routine

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the subroutine that is to be placed in line 4ØØØ. This is just our familiar INKEY$ routine, along with a bit of PRINT, so we can deal with it right away. We might have a routine just like this noted down somewhere. If we don't have it on record, we will have to write it, and Fig. 11.4 shows the form it might take. The subroutine is straightforward, and that's why we can deal with it right away! Type it in, and now test the core program with this subroutine in place.

```
4000 PRINT "Would you like another one.
?": PRINT "Please type Y or N. "
4010 LET K$=INKEY$: IF K$="" THEN  GOTO
 4010
4020 RETURN
```

Fig. 11.4. The subroutine for line 4ØØØ.

The hard part

Now we come to what you might think is the hardest part of the job – the subroutine which carries out the Play action. In fact, you don't have to learn anything new to do this. The Play subroutine is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a subroutine to be dealt with later.

As an example, take a look at Fig. 11.5. This is a plan for the Play subroutine, which also includes information that we shall need for the setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not

1. Keep the answers as an array of strings of ASCII codes. Use upper-case codes, each of which consists of two digits.
2. Keep the list of countries as another string array.
3. Pick a number at random for array item. The number which selects the country also selects the correct answer.
4. Use variable TR to record tries, and SC to keep score.
5. Use variable GO to record the number of attempts at one question.

Fig. 11.5. Planning the 'Play' subroutine.

cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters. I've decided to put one answer in each DATA line in the form of ASCII codes, using upper-cases codes. Each code for an upper-case letter consists of two digits, so we can run the codes together. We can pick out the codes by using a MID$ instruction that takes the numbers two digits at a time. That's the first item for this subroutine.

The next one is that we shall keep the names of the countries, and the ASCII codes for the capitals, in two string arrays. This has several advantages. One of them is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are 1 to 1Ø, then we can keep the answers in another array which has corresponding subscript numbers. The subroutine can easily find the correct string of numbers, and then pick the codes out of it. Another possibility would be to keep each answer in a separate DATA line, and select by using RESTORE. If, for example, we kept the answer for Question 1 in line 9ØØ1, and the answer for Question 2 in line 9ØØ2, and so on, we could use RESTORE 9ØØØ + V to find the line that contained the correct answer. For this example, though, we'll stick to the use of an array.

The next thing that the plan settles is the names that we shall use for the variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using 'SC' for the score and 'TR' for the number of tries looks self-explanatory. The third one, 'GO' is one that we shall use to count how many times one question is attempted. We shall use TRUE$ for the correct answer to each question. Finally, we decide on names for the arrays that will hold the country names and the capital city codes – Q$ () and A$ ().

Play for today

Figure 11.6 shows what I've ended up with as a result of the plan in Fig. 11.5. The steps are to pick a random number, use it to print a country name, and then find the answer. That's all, because the

```
2000 LET GO=0: LET V=INT(1+RND*10)
2001 REM Pick from ten at random
2010 CLS : CSR 5,8: PRINT "The country
is ";Q$(V)
2020 CSR 5,10: PRINT "The capital is -
"
2030 CSR 5,12: INPUT "(Please answer he
re) - ";X$: LET TR=TR+1
2040 GOSUB 5000
2041 REM Find correct answer.
2050 RETURN
```

Fig. 11.6. The program lines for the 'Play' subroutine.

checking of the answer and the scoring is dealt with by another subroutine. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time. As it is, I've had to put another subroutine into this one to keep things short.

We start in line 2ØØØ with setting variable GO to zero. This will be done earlier, and at other places in the program, but it must be done here in particular. GO is the variable that we use to count the number of shots at an answer, and it has to start at zero each time a new country name is displayed. The second part of line 2ØØØ then picks a number, at random, lying between 1 and 1Ø. As before, we use line 2ØØ1 to hold a REM that reminds us of what's going on. Lines 2Ø1Ø to 2Ø3Ø are straightforward stuff. We print the name of the country that corresponds to the random number, and ask for an answer, the capital city of that country. The last section of line 2Ø3Ø counts the number of attempts. This is the logical place to put this step, because we want to increment the count each time there is an answer. Now it's chicken-out time. I don't want to get involved in the reading of ASCII codes right now, so I'll leave it to a subroutine, starting in line 5ØØØ, which I'll write later. The REM in line 2Ø41 reminds me what this new subroutine will have to do, and the Play subroutine ends with the usual RETURN in line 2Ø5Ø.

Down among the details

With the Play subroutine safely on tape, we can think now about the details. The first one to look at should be one that precedes or follows the Play step, and I've chosen the Score routine. As usual, it has to be planned, and Fig. 11.7 shows the plan. Each time that there is a correct answer, the number variable 'SC' will be incremented, and we can go back to the main program. More is needed if the

1. For a correct answer, increment SC
2. For an incorrect answer, with GO = Ø, allow another try, and make GO = 1
3. For second incorrect answer, with GO = 1, pass on to the next question, and make GO = Ø again.

Fig. 11.7. Planning the 'Score' subroutine.

answer does not match exactly. We need to print a message, and allow another go. If the result of this next go is not correct, that's an end to the attempts.

Figure 11.8 shows the program subroutine developed from this plan. Line 3ØØØ deals with a correct answer. The GOTO 3Ø3Ø

```
3000 PRINT : IF X$=TRUE$ THEN  LET SC=S
C+1: PRINT "Correct - your score is now
 ";SC;" in ";TR;" attempts.": GOSUB 700
0: GOTO 3030
3010 IF GO=0 THEN  PRINT "Not correct -
 but it might be your ": PRINT "spellin
g! You get another shot free.": GOSUB 7
000: LET GO=1: GOSUB 2010: GOTO 3000
3020 LET GO=0: PRINT "No luck - try the
 next one."
3030 RETURN
```

Fig. 11.8. The 'Score' subroutine written.

ensures that if the answer was correct, the rest of the subroutine is skipped, and the subroutine returns. There is a time delay before returning, and this is ensured by the subroutine at line 7ØØØ. We'll leave that one until later also. If the answer is not correct, though, line 3Ø1Ø swings into action. This prints a message, and then calls the subroutine at line 2Ø1Ø again so that the user can make another answer entry. The GOTO 3ØØØ at the end of line 3Ø1Ø then tests this answer again.

Now there's a piece of cunning here. The number variable 'GO' must start with a value of ∅ – we have already made sure of this. When there is an incorrect answer, however, and 'GO' is still ∅, line 3∅1∅ is carried out. One of the actions of line 3∅1∅, however, is to set 'GO' to 1. When you answer again, with GO = 1, line 3∅∅∅ will be used, and if your second answer is wrong, line 3∅1∅ cannot be used, because 'GO' is not zero. The next line that is tried, then, is 3∅2∅. This puts 'GO' back to zero for the next round, prints a sympathetic message, pauses, and then lets the subroutine return in line 3∅3∅.

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 11.9 shows the subroutine that deals with dimensioning and arrays. Line 14∅∅ sets all the variables for the

```
1400 LET TR=0: LET SC=0: LET GO=0
1401 REM all variables to zero
1410 DIM Q$(10,20),A$(10,22)
1411 REM list
1420 FOR J=1 TO 10: READ Q$(J): NEXT
1430 FOR J=1 TO 10: READ A$(J): NEXT
1440 RETURN
```

Fig. 11.9. The dimensioning and array subroutine.

scoring system to zero. Line 141∅ dimensions the arrays Q$ and A$ that will be used for the names of the countries and the answers. Line 142∅ then reads the names from a data list into the array. Line 143∅ reads the strings of numbers, which provide the answers, into the array A$ – and that's it! We can write the DATA lines later, as usual.

Next comes the business of finding the answer. We have planned this, so it shouldn't need too much hassle. Figure 11.10 shows the program lines. The variable V is the one that we have selected at

```
5000 LET TRUE$="": LET X=2
5010 LET L=VAL(MID$(A$(V),X,2))
5020 IF L>=65 THEN  LET TRUE$=TRUE$+CHR
$(L): LET X=X+2: GOTO 5010
5030 RETURN
```

Fig. 11.10. Checking the answer.

random, and it's used to select a member of the A$ array, A$(V). We want to pick out numbers, two digits at a time from this string, but only for as many numbers as are in the string. Now there are complications here that we need to understand. One is that the Memotech automatically places a space ahead of any number or string which is at the start of a DATA line. This means that we will

have to be rather careful about any string slicing that we do. The other point is that each string in the array A$ is 22 characters long! Now the strings of numbers are not of this length, but because we have dimensioned for 22 characters, this is what we get!

Incidentally, you'll find that the longest string of digits is 20 characters long, but you can't get away with this as a dimension, because of that extra (invisible) space. Line 5ØØØ gets things started by setting the answer, TRUE$ to a blank, and a variable X to 2. X is the variable that we shall use for slicing each string in the array, and the reason that we have to set it to 2 rather than to 1 is because of the space ahead of each string in the array. This applies only when we have used READ...DATA, incidentally, not when strings have been allocated using INPUT. Line 5Ø1Ø then finds the number which results from slicing two digits from the string. VAL has to be used

```
P2
                    INSTRUCTIONS

 You will be presented with the name
of a country. Make sure that the ALPHA
 LOCK key is set for capitals, and then
 type the name of the CAPITAL CITY of
the country. You can edit this as you
like, and when it looks right, press
the RET key.
 If your answer is wrong, which may be
because you used small letters or
spelled the name incorrectly, you will
be told. You get another chance at the
same country - but no more.
 The computer will keep your score,
and print it each time.


  NOW PRESS THE RET KEY TO START.
```

(a)

```
X1   *D P1.
     *E
     *R
```

(b)

Fig. 11.11. The instructions - always leave these until you have almost finished. (a) The NODDY text, (b) the control program.

here so as to convert the string of two digits into a two digit *number*. I have used MID$ for slicing, because the alternative slicing method does not appear to operate satisfactorily for arrays. The next line, 5Ø2Ø, tests this number L. This is important, because this is how we know when we have reached the end of the real string, which may have only ten characters, not 22. If the next part of line 5Ø2Ø is carried out with empty pieces of string (blank spaces), this will cause an error message later when we compare TRUE$ to the answer that you have typed. If L>= 65 (meaning if it's an ASCII code for a letter) then the next part of line 5Ø2Ø builds up the answer string, which we call TRUE$. This was set to a blank in the first part of line 5ØØØ to ensure that we always start with a blank string, not with the previous answer. When this character has been added to TRUE$, we then increment X by 2 (because we pick two characters at a time), and go back to 5Ø1Ø for the next letter. This is another reason for dimensioning the array A$ generously. We must have at least one space following each code string. If we don't, there will be nothing to stop the loop back at line 5Ø2Ø going on for too many times. What stops this loop is finding a value of L which is less than 65. Another way that you could ensure this would be by placing a full stop following each item in the array. When this loop is finished, we shall have the letters of the answer present in TRUE$. That's the hard work over. Fig. 11.11(a) shows the NODDY text for the instructions, and Fig. 11.11(b) shows the control program for this page. Figure 11.12 is the NODDY title subroutine. Each of these

```
THE COUNTRY GAME


(Press RET when you are ready)
```

(a)

```
X2    *D P2.
      *E
      *R
```

(b)

Fig. 11.12. The title. (a) NODDY text, (b) control program.

NODDY control programs includes a 'press RET' step so that you can take your time reading the text. Finally, Fig. 11.13 shows the DATA lines and the subroutine which is used to provide a delay after each answer.

```
5500 DATA   France,Austria,Colombia,Den
mark,India,Italy,Holland,Portugal,Venez
uela,Yugoslavia
6000 DATA  8065827383
6001 DATA  867369787865
6002 DATA  667971798465
6003 DATA  677980697872657l6978
6004 DATA  6869767273
6005 DATA  82797769
6006 DATA  6577838469826865 77
6007 DATA  767383667978
6008 DATA  67658265676583
6009 DATA  66697671826568 69
7000 PAUSE 1000
7010 RETURN
```

Fig. 11.13. The DATA lines that are needed, along with a time delay subroutine.

Now we can put it all together, and try it out. Because its been designed in sections like this, it's easy for you to modify it. You can use different DATA, for example. You can use a lot more data – but remember to change the DIM statements in line 141Ø. You can, of course, make this a question-and-answer game on something entirely different, just by changing the data and the instructions. It's useful, however, to keep the game as it is for a time, and work on it in a different way. This should be concerned with improving the mechanism of the game. This is never something that you can plan adequately in advance unless you have had a lot of experience in game design and playing. Without this experience, you need to have played the game to appreciate in what respects it is inadequate. Only then can you make a list of these flaws, and decide what you can do about them.

Take a look at the flaws in this program. There are too few items, to start with. That's easy to deal with. The next flaw is more serious. The action of picking numbers at random can mean that an item is repeated, perhaps twice in a row. Can you arrange things so as to avoid this? It should be fairly simple to ensure that the program does not pick the same number twice running. Just allocate the random number V to another variable Z before you finish using V. The next

time a value of V is picked, compare it to Z, and if the two are the same, repeat the random number step. A more difficult exercise is to ensure that the same number is *never* picked again. The classic way of doing this is to rearrange the array each time a number has been picked. Suppose, for example, that we have ten items, and that we pick item 3. When we have done the question and answer, then we have to decide what to do with this item. If the question has not been answered correctly, nothing is done – the question can remain. If it has been answered correctly, however, we can get rid of it. We do this by swopping item 3 with item 1Ø, so that what was item 3 is now item 1Ø, and what was item 1Ø is item 3. Once this has been done, the line that generates the random number is set so that it picks from 9 rather than from ten. You do this by using a variable in place of the 1Ø in line 2ØØØ.

There are other types of improvements. Are you happy with the presentation? Could you use different virtual screens for the question and answer? What about putting a time limit on the answer? The Memotech has CLOCK and TIME$ variables (well explained in the manual) which will let you print times, or you can use PAUSE to generate delays. What about using one colour of text for questions and another for answers? Could you make an incorrect answer word flash? There is no limit to what you can think of and do to build on a framework like this. You have now reached the launching-pad stage in the use of your Memotech. From now on, you'll find that you will be picking up useful information from the monthly magazines rather than from books. Of all the range of monthly magazines, *Personal Computer World* is the one that contains the greatest number of useful hints and tips when you have some programming experience. You will also find *Practical Computing* useful, however, for news of business applications, and *Your Computer* useful if you are interested in games. Good luck!

Appendix A
Cassette Head Adjustment

Cassette recorders, like open-reel tape recorders, work on the principle of pulling plastic tape, which has been coated with magnetic material, past a 'tapehead', which is a miniature electromagnet. The important part of any tapehead is the 'gap', a tiny slit in the metal, too fine to see except under a microscope. This slit should be placed so that it is at 90° to the direction of movement of the tape, but this angle, which can be adjusted by tilting the whole tapehead, is seldom precisely set, even when the recorder has been quite expensive. A poorly set-up head will make it difficult to load programs that have been recorded on correctly set-up equipment

CASSETTE RECORDER HEAD ALIGNMENT METHOD

(1) Insert a cassette, with a long program, into the recorder.
(2) Remove cable connections between the computer and the recorder.
(3) Start playing the cassette. Set the volume control to a comfortable level, and listen. Any tone control should be set to give maximum treble.
(4) Insert a thin-bladed screwdriver into the head-alignment screw-head. On some recorders this is reached with the cassette flap shut, through a hole in the casing. On other models, it will be necessary to open the flap. This may have to be done *before* playing the tape.
(5) Adjust the azimuth screw *slightly* in each direction, listening to increase in the treble (a sharper sound). If adjustment causes the note to sound more muffled, reverse the direction of turning. Adjust until the note is at its sharpest.
(6) Rewind the cassette, and make the connections between the computer and the recorder.
(7) Try to load a program. If good loading cannot be achieved, repeat the procedure, but look for *another* setting which produces maximum treble.
(8) NOTE that this procedure is needed only if a tape from a reputable source cannot be loaded. Tapes made on a recorder will be loaded by that recorder unless there is a serious fault. Once the adjustment described above has been carried out, tapes recorded *before* the adjustment may not load correctly *after* the adjustment.

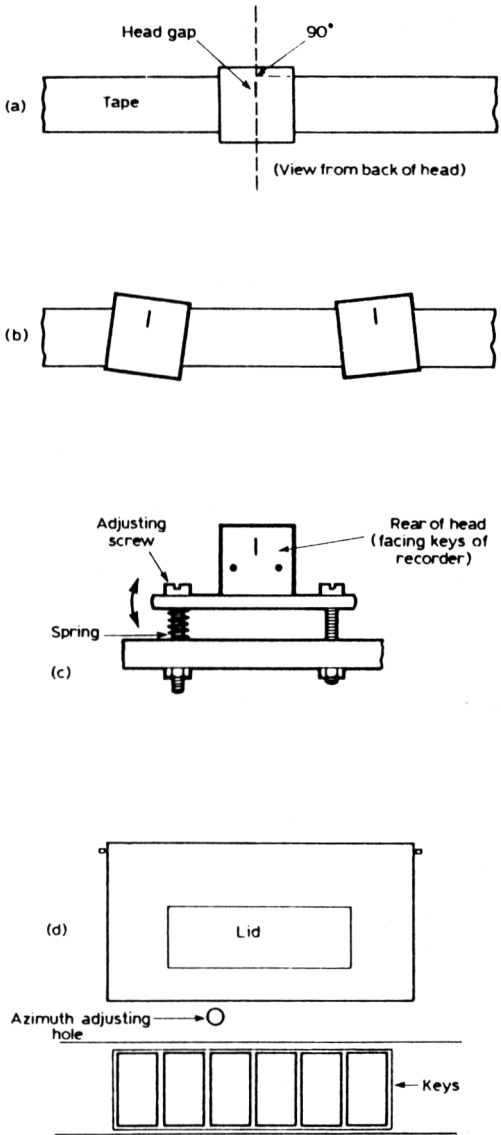


Fig. A1. Tape-head azimuth. The narrow slit in the tape-head (a) is normally at 90° to the edge of the tape. This is the correct azimuth angle, but a surprising number of recorders have this maladjusted. Any deviation from this angle (b) causes muffled sound and poor loading. The angle can be altered by turning an adjusting screw (c) which is on the head mounting. This is often reached through a hole in the casing of the recorder (d). (Courtesy of Keith Dickson Publishing.)

(bought software, for example), though you will always be able to load tapes which have been saved on the same equipment with the same head adjustment. NEVER touch the recording head with anything metal – but you can set the alignment fairly easily, following the scheme outlined here, in Fig. A1.

After a lot of use, it's important to clean the head. Use a cleaning kit, such as the BIB, and follow the instructions carefully. A few authorities say that the head should be demagnetised at intervals, but in five years of using the same recorder for computer loading and saving, I have never found this to be necessary.

Appendix B
Editing

Editing means changing something that has already appeared on the screen. Any feature of a line, including its line number can be changed by editing. The editing process can be carried out:

(a) While a line is being entered, before RET has been pressed.
(b) After pressing RET, if there is an error message and the line is not entered.
(c) At a later stage, after a line has been entered, but before the program is run.
(d) When an error is signalled during running.

Dealing with these in order:
While a line is being entered, all of the editing commands, below, can be used. Editing is completed by pressing the RET key.

If there is an error message after pressing RET, the line is not entered into the memory. It is automatically prepared for editing, with the cursor over the line number or near the mistake.

When the line has been entered, but the program has not been run, you must prepare for editing by typing EDIT 2ØØ (using the correct line number), and then pressing RET.

When the program stops with an error message, the line in which an error has been traced will be put on to the editing screen, with the cursor on its line number or over the error. This does not always mean that there is an error in this line. For example, if the line contains LET X= A*2, and A has not been declared (given a value), an error will be signalled in this line, because this is the first mention of the variable A. You don't need to edit this line, so simply press RET. Then add a line which creates a value for A.

Editing commands

(1) Cursor keys. The keys on the 12-key pad are used for editing.

The keys which are marked with the right-arrow and the left-arrow will move the cursor in the directions indicated. The up-arrow and down-arrow keys have no effect – they are used for NODDY and for DSI commands. You can move the cursor to the error using just the two keys.

(2) To replace a letter, simply place the cursor over the error and type the correct letter.

(3) To delete a letter, place the cursor over it and press DEL (on the 6 key).

(4) To insert a letter, place the cursor over the letter that will follow the insertion. Press INS, and then type the letter. You can then make other insertions without pressing INS again. Pressing INS again stops insertion. Insertion is also stopped when you press RET.

(5) Pressing EOL will delete everything in a line to the right of the cursor. 'Line' in this sense means a complete numbered BASIC line, which may take several lines on the screen.

(6) The HOME key places the cursor at the start of the line.

(7) The TAB key moves the cursor eight spaces to the right.

(8) Pressing RET ends editing, and places the line into memory – unless another error is detected!

Editing oddities

(1) The ENT key (over CLS) can be used in place of RET, and will accept a line even if there is an error. The RET key will automatically switch to editing if an error is detected.

(2) CLS is used to get out of an unwanted line. If you type : 1Ø LPRINT and then decide not to use this, pressing CLS, then RET, will delete it. It's also a useful way of getting rid of a line with a mistake which you can't correct!

(3) When an error is discovered in a program, and the program stops with an error message, the computer automatically switches to editing the 'faulty' line. As explained earlier, the line may not be faulty. In addition, if you edit, press RET or press ENT, all variables will be deleted, and you can't print variable values to find out what went wrong. In such a case, press CLS, then RET, and then print

your variables, using commands like P.A : P.B$ and so on. If the mistake is obvious, of course, then just edit it.

(4) The AUTO feature can be useful. If you type AUTO 1Ø,1Ø (or press the F4 key followed by 1Ø,1Ø), the computer will generate line numbers for you each time you press RET. To stop the auto-numbering, press BRK. This can also be used for deleting a number of lines. Suppose we want to delete lines 1ØØ to 15Ø inclusive, and the line numbers increase in tens. Type AUTO 1ØØ,1Ø (or use F4 1ØØ,1Ø), and press RET as each number comes up. When you reach a number that you don't want to delete, press CLS, then RET.

Appendix C
The Programmed Keys

The eight keys of the set at the right-hand side of the keyboard are marked F1 to F8. They are referred to as *function* (hence the 'F') or programmed keys, and their normal action is to provide the keywords that are shown below. A different set of keywords is provided when the SHIFT key is pressed at the same time. The keyword is not visible on the screen until RET is pressed, and if the keyword requires numbers (like AUTO, for example), then these must be typed *before* RET is pressed. As the manual indicates, these keys can also be used for other purposes by making use of ASC (INKEY$), since each key will give a different number code.

Key functions

Key alone	*KEY*	*Key plus SHIFT*
REM	F1	CRVS
CLS	F2	CLEAR
ASSEM	F3	CLOCK
AUTO	F4	ATTR
BAUD	F5	COLOUR
VS	F6	INK
CONT	F7	CSR
USER	F8	DATA

Appendix D

Assembler and Machine Code

BASIC is an *interpreted* computer language. Interpreted means that each BASIC keyword is used to carry out a sequence of much simpler commands to the microprocessor chip which handles most of the computer's actions. The microprocessor is controlled by number codes, which are called *machine code*. Each different type of microprocessor uses a different type of machine code; in the case of the Memotech machines, the microprocessor is a Z-80.

By discarding BASIC, and taking direct control of the Z-80 with machine code, you can achieve much faster graphics, better control of sound, and effects (like making the programmed keys give different keywords) that are not possible when you use only BASIC. Machine code, however, is a language which needs a book to itself. For many computers, it is necessary to buy a program which is called an assembler. This allows you to create machine code in a reasonably simple way, by using commands which are words rather than just numbers. The Memotech computers have their assembler built in, so that you can progress to machine code whenever you are ready for it – watch out for the book when the time comes!

Appendix E

Other Trigonometrical Functions

The Memotech uses only SIN, COS TAN and ATN functions as standard. If you want to use the Memotech for other trigonometrical actions, you will have to apply the formulae below.

Name	*Expression*
Secant	SEC (X) = 1/COS (X)
Cosecant	CSC (X) = 1/SIN (X)
Cotangent	COT(X) = 1/TAN (X)
Arcsine	ASN (X) = ATN (X/SQR (−X*X+1))
Arccos	ACS (X)=−ATN (X/SQR (−X*X+1))+1.5708
Arcsec	ARCSEC (X)= ATN (SQR (X*X−1))+ (SGN (X) −1)*1.5708
Arc-cosec	ARCCSC (X)= ATN (1/SQR (X*X−1))+ (SGN (X) −1)*1.5708
Arc-cot	ARCCOT (X)= −ATN (X) + 1.5708

Index

HOW TO MASTER THE MEMOTECH!

These splendid machines offer spectacular colour, graphics and sound. The hardware is impressive and is much better equipped for add-ons than most other machines of similar price.

Naturally, with all this power at your finger tips, you will want to achieve results fast. Ian Sinclair does not waste time with unnecessary theory. Very many short, practical examples are provided throughout for you to try out and enjoy as you learn to write your own programs and create your own special effects. This essential book for all users guides you step-by-step from first principles until you are set firmly on an expert course. It will also serve you as a user's reference for a long time to come.

The Author

Ian Sinclair is a well known contributor to journals such as *Personal Computer World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over fifty books on aspects of electronics and computing, mainly aimed at the beginner.

Also from Granada

THE MEMOTECH GAMES BOOK
Owen Bishop and Audrey Bishop
0 246 12407 5

Front cover illustration by Peter Goodfellow

GRANADA PUBLISHING
Printed in Great Britain 0 246 12408 3

£6.95 net